The jsonlite Package: A Practical and Consistent Mapping Between JSON Data and R Objects

Jeroen Ooms

October 20, 2014

Abstract

A naive realization of JSON data in R maps JSON *arrays* to an unnamed list, and JSON *objects* to a named list. However, in practice a list is an awkward, inefficient type to store and manipulate data. Most statistical applications work with (homogeneous) vectors, matrices or data frames. Therefore JSON packages in R typically define certain special cases of JSON structures which map to simpler R types. Currently no formal guidelines or consensus exists on how R data 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 edge cases. This paper explicitly describes a mapping between R classes and JSON data, highlights potential problems, and outlines conventions that generalize the mapping to cover all common structures. We emphasize the importance of type consistency when using JSON to exchange dynamic data, and illustrate using examples and anecdotes. The jsonlite package is used throughout the paper as a reference implementation.

1 Introduction

JavaScript Object Notation (JSON) is a text format for the serialization of structured data (Crockford, 2006a). It is derived from the object literals of JavaScript, as defined in the ECMAScript programming language standard (Ecma International, 1999). Design of JSON is simple and concise in comparison with other text based formats, and it was originally proposed by Douglas Crockford as a "fat-free alternative to XML" (Crockford, 2006b). The syntax is easy for humans to read and write, easy for machines to parse and generate and completely described in a single page at http://www.json.org. The character encoding of JSON text is always Unicode, using UTF-8 by default (Crockford, 2006a), making it naturally compatible with non-latin alphabets. Over the past years, JSON has become hugely popular on the internet as a general purpose data interchange format. High quality parsing libraries are available for almost any programming language, making it easy to implement systems and applications that exchange data over the network using JSON. For R (R Core Team, 2014), several packages that assist the user in generating, parsing and validating JSON are available through CRAN, including rjson (Couture-Beil, 2013), RJSONIO (Temple Lang, 2013), and jsonlite (Ooms et al., 2014).

The emphasis of this paper is not on discussing the JSON format or any particular implementation for using JSON with R. We refer to Nolan and Temple Lang (2014) for a comprehensive introduction, or one of the many tutorials available on the web. Instead we take a high level view and discuss how R data structures are most naturally represented in JSON. This is not a trivial problem, particularly for complex or relational data as they frequently appear in statistical applications. Several R packages implement toJSON and fromJSON functions which directly convert R objects into JSON and vice versa. However, the exact mapping between the various R data classes JSON structures is not self evident. Currently, there are no formal guidelines, or even consensus between implementations on how R data 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 certain special cases. Furthermore, best practices of representing data in JSON have been established outside the R community. Incorporating these conventions where possible is important to maximize interoperability.

1.1 Parsing and type safety

The JSON format specifies 4 primitive types (**string**, **number**, **boolean**, **null**) and two *universal structures*:

- A JSON *object*: an unordered collection of zero or more name-value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array.
- A JSON *array*: an ordered sequence of zero or more values.

Both these structures are heterogeneous; i.e. they are allowed to contain elements of different types. Therefore, the native R realization of these structures is a **named list** for JSON objects, and **unnamed list** for JSON arrays. However, in practice a list is an awkward, inefficient type to store and manipulate data in R. Most statistical applications work with (homogeneous) vectors, matrices or data frames. In order to give these data structures a JSON representation, we can define certain special cases of JSON structures which get parsed into other, more specific R types. For example, one convention which all current implementations have in common is that a homogeneous array of primitives gets parsed into an **atomic vector** instead of a **list**. The RJSONIO documentation uses the term "simplify" for this behavior, and we adopt this jargon.

```
txt <- '[12, 3, 7]'
x <- fromJSON(txt)
is(x)
```

```
[1] "integer"             "numeric"             "vector"
[4] "data.frameRowLabels"
```

```
print(x)
```

```
[1] 12  3  7
```

This seems very reasonable and it is the only practical solution to represent vectors in JSON. However the price we pay is that automatic simplification can compromise type-safety in the context of dynamic data. For example, suppose an R package uses fromJSON to pull data from a JSON API on the web and that for

some particular combination of parameters the result includes a `null` value, e.g: `[12, null, 7]`. This is actually quite common, many `API`'s use `null` for missing values or unset fields. This case makes the behavior of parser ambiguous, because the `JSON` array is technically no longer homogeneous. And indeed, some implementations will now return a `list` instead of a `vector`. If the user had not anticipated this scenario and the script assumes a `vector`, the code is likely to run into type errors.

The lesson here is that we need to be very specific and explicit about the mapping that is implemented to convert between `JSON` data and `R` objects. When relying on `JSON` as a data interchange format, the behavior of the parser 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 content of the data is dynamic. Similarly, `R` code using dynamic `JSON` data from an external source is only reliable when the conversion from `JSON` to `R` is consistent. Moreover a practical mapping must incorporate existing conventions and use the most natural representation of certain structures in `R`. In the example above, we could argue that instead of falling back on a `list`, the array is more naturally interpreted as a numeric vector where the `null` becomes a missing value (`NA`). These principles will extrapolate as we start discussing more complex `JSON` structures representing matrices and data frames.

1.2 Reference implementation: the `jsonlite` package

The `jsonlite` package provides a reference implementation of the conventions proposed in this document. It is a fork of the `RJSONIO` package by Duncan Temple Lang, which builds on `libjson C++` library from Jonathan Wallace. `jsonlite` 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 conventions.

```
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.

1.3 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 method dispatch. For all of the common classes in `R`, the `jsonlite` package implements `toJSON` methods as described in this document. Users in `R` can extend this system by implementing additional methods for other classes. 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 class-based method dispatch 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 resulting JSON closely resembles the internal structure of the underlying C data types, and can be perfectly restored to the original R object using `unserializeJSON`. This system is relatively straightforward to implement, but 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 data 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.

1.4 Scope and limitations

Before continuing, we want to stress some limitations of encoding R data structures in JSON. Most importantly, there are limitations to the 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 (Chodorow, 2013) or MSGPACK (Furuhashi, 2014), 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, we can encode a blob as a string using `base64`.

Finally, as mentioned earlier, `fromJSON` is not a perfect inverse function of `toJSON`, as is the case for `serialializeJSON` and `unserializeJSON`. The class based mappings are designed for concise and practical

encoding of the various common data structures. Our implementation of `toJSON` and `fromJSON` approximates 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 coerced 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 consumer side.

2 Converting between `JSON` data and `R` classes

This section lists examples of how the common `R` classes are represented in `JSON`. As explained before, the `toJSON` function relies on method dispatch, which means that objects get encoded according to their `class` attribute. 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. Atomic vectors hold an ordered, homogeneous set of values of type `logical` (booleans), `character` (strings), `raw` (bytes), `numeric` (doubles), `complex` (complex numbers with a real and imaginary part), or `integer`. 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 map to `JSON` arrays:

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

```
[1,2,3.1416]
```

The `JSON` array is the only appropriate structure to encode a vector, even though 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 values: a concept foreign to many other languages. Besides regular values, each vector type in `R` 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`. The `JSON` format 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)
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)
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 it is impossible to distinguish between different types of missing data, which could be a problem for numeric vectors. The values `Inf`, `-Inf`, `NA` and `NaN` carry different meanings, and these should not get lost in the encoding. 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 string 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:

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

```
[true,null,null,false]
```

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

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

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

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

```
#Non-default behavior
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`):

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

```
["2014-10-20 13:47:30","2014-10-20 13:47:31","2014-10-20 13:47:32"]
```

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

```
["2014-10-21","2014-10-22","2014-10-23"]
```

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

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

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

```
["0.7039-0.5843i","0.1294-0.9719i","0.8479-0.2835i"]
```

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` function, 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
toJSON(vector())
```

```
[]
```

```
toJSON(pi)
```

```
[3.1416]
```

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

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

```
toJSON(list(foo=pi))
```

```
{"foo":[3.1416]}
```

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

```
[[]]
```

```
toJSON(list(pi))
```

```
[[3.1416]]
```

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 dataset this seems 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 as a `JSON` array. The client code then parses the `JSON`, and 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 suddenly returns a primitive value where the client is expecting an array, the application will likely break. Therefore, 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. To avoid this, `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 subprograms (Lawson et al., 1979) such as `LAPACK` (Anderson et al., 1987). 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, which are in turn building blocks of 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 rows and columns 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, the way it is printed and indexed makes it conceptually a 2 dimensional structure. In `jsonlite` a matrix maps to an array of equal-length subarrays:

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

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

We expect this representation will be the most intuitive to interpret, also within languages that do not have a native notion of a matrix. Note that even though `R` stores matrices in *column major* order, `jsonlite` encodes matrices in *row major* order. This is a more conventional and intuitive way to represent matrices and is consistent with the row-based encoding of data frames discussed in the next section. When the `JSON` string is properly indented (recall that white space and line breaks 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)
toJSON(x)

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

toJSON(x, na="null")

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

toJSON(matrix(pi))

[[3.1416]]
```

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` mapping for matrices for several reasons. First of all, because this attribute

is optional, either row or column names or both could be `NULL`. This makes it difficult to define a practical mapping 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. The linear algebra subroutines mentioned before completely ignore them, and never include any names in their output. So there is often little purpose of setting names in the first place, other than annotation.

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. Wickham (2014) 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"* as the most common source of untidy data. This often happens when the structure is optimized for presentation (e.g. printing), rather than computation. In the following example taken from Wickham, the predictor variable (treatment) is stored in the column headers rather than the actual data. As a result, these values do not get included in the `JSON` output:

```
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
```

```
toJSON(x)
```

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

Wickham 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
```

```
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 (described later) is more suitable when we want to refer to rows or fields by their name. Any matrix can easily be converted to a data-frame using the `as.data.frame` function:

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

```
[
    {
        "Treatment B": 5,
        "_row": "Joe"
    },
```

```
    {
        "Treatment A": 1,
        "_row": "Jane"
    },
    {
        "Treatment A": 2,
        "Treatment B": 3,
        "_row": "Mary"
    }
]
```

For some cases this results in the desired output, 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 maps to a `JSON` array:

```
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 from their contents: an `R` vector results in a `JSON` array containing only primitives, whereas a list results in a `JSON` array containing 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] FALSE
```

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` maps to a `JSON` *object*:

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

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

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

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

```
{"foo":{"bar":{"baz":[3.1416]}}}
```

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

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 is given 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 *rows*. 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` was quite similar to the other available packages that implement `toJSON` and `fromJSON` 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 for compatibility with conventional ways of encoding table-like structures outside the `R` community. The implementation is more involved, but results in a powerful and more natural way of representing data frames in `JSON`.

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 fashion. A column based structure consists of a named collection of equal-length, homogeneous 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 these 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 stitched back together. For row-based implementations, it is the exact other way around: we can easily manipulate a particular record, but to insert/extract a whole column we would need to iterate over all records in the table and read/modify the appropriate field in each of them.

The data frame class in `R` is implemented in a column based fashion: 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`.

Unfortunately `R` is an exception in its preference for column-based storage: most languages, systems, databases, `API`'s, 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 on the web, into column-based data frames which are convenient for statistical computing.

2.4.2 Row based data frame encoding

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` maps data frames by default to an array of records:

```
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 `JSON` primitives, whereas lists always contain `JSON` objects or arrays:

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

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

This leads to the following convention: when encoding `R` objects, `JSON` primitives only appear in vectors and data-frame rows. Primitives within a `JSON` array indicate a vector, and 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` representation, 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
```

```
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 method used on the web, and we expect this encoding will most likely lead to the correct interpretation of *missingness*, even in languages without an 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. The `jsonlite` implementation maps these subrecords to 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
```

```
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] FALSE
```

When encountering `JSON` data containing nested records on the web, 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 vehicles, 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.

```
y <- fromJSON(myjson, flatten=TRUE)
str(y)
```

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

```
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 separate 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()
)
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 nested data frames contain the same fields, as would be the case in an actual relational data base.

3 Structural consistency and type safety in 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 definition. In other languages such as `S3` or `JavaScript`, the fields are not enforced by the language but rather at the discretion of the programmer. One way or another they assume that members of a certain class agree on 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 exactly the same way as for 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 template 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 can be interpreted and processed appropriately with no exceptions during run-time.

Some 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 relatively uncommon. Some initiatives for `JSON` schema languages exist (Galiegue and Zyp, 2013), but they are not 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 type safe.

3.2 Rule 1: Fixed keys

When using `JSON` without a schema, there are no restrictions on the keys (field names) that can appear in a particular object. However, a source of data that returns a different set of keys 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 separates the data from metadata. At run time, records can be inserted or deleted, and a certain query might return different content 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 separated 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 in practice was given by Mike Dewar at the New York Open Statistical Programming Meetup on Jan. 12, 2012 (Dewar, 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" : 124 },
  { "1325344456" : 131 },
  { "1325344478" : 137 }
]
```

Even though being valid JSON, dynamic keys as in the example above are likely to introduce trouble. Most software will have great difficulty processing these values if we can not specify the keys in the code. Moreover when documenting the API, either informally or formally using a schema language, we need to describe for each property in the data what the value means and is composed of. Thereby a client or consumer can 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 is difficult to extend 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": 124 },
  { "time": "1325344456" : "price": 131 },
  { "time": "1325344478" : "price": 137 }
]
```

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 properties can be added to each observation without breaking backward compatibility.

In the context of R, consistency 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, 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 (2014).

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))
toJSON(x)
```

```
[["FOO"],[1,2,3],{"bar":[3.1416]}]
```

However even though it is possible to generate such JSON, it is bad practice. Fields or collections with ambiguous object types are difficult 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*. In dynamically typed languages, the programmer needs to make sure that properties are of the correct type before encoding into JSON. For R, this means that the `unnamed lists` type is best avoided when designing interoperable structures because this type is not homogeneous.

Note that consistency is somewhat subjective as it refers to the *meaning* of the elements; they do not necessarily have precisely the same structure. What is important is to keep in mind that the consumer of the data can interpret and process 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"))
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, 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. A collection of different object classes should be separated and organized using a named list:

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

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

```
}
```

This might seem obvious, but dynamic languages such as R can make it dangerously tempting to generate data containing mixed-type collections. Such inconsistent typing makes it very difficult to consume the data and creates a likely source of nasty bugs. Using consistent field names/types and homogeneous JSON arrays is a strong convention among public JSON API's, for good reasons. We recommend R users to respect these conventions when generating JSON data in R.

References

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (Software, Environments and Tools)*. Society for Industrial and Applied Mathematics, 3 edition, 1 1987. ISBN 9780898714470. URL http://amazon.com/o/ASIN/0898714478/.

Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, second edition, 5 2013. ISBN 9781449344689. URL http://amazon.com/o/ASIN/1449344682/.

Alex Couture-Beil. *rjson: JSON for R*, 2013. URL http://CRAN.R-project.org/package=rjson. R package version 0.2.13.

D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006a. URL http://www.ietf.org/rfc/rfc4627.txt. Obsoleted by RFCs 7158, 7159.

Douglas Crockford. JSON: The Fat-free Alternative to XML. In *Proceedings of XML*, volume 2006, 2006b. URL http://www.json.org/fatfree.html.

Mike Dewar. First steps in data visualisation using d3.js, 2012. URL http://vimeo.com/35005701#t=7m17s. New York Open Statistical Programming Meetup on Jan. 12, 2012.

Ecma International. ECMAScript Language Specification. *European Association for Standardizing Information and Communication Systems*, 1999. URL http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

Sadayuki Furuhashi. *MessagePack: It's Like JSON. But Fast and Small*, 2014. URL http://msgpack.org/.

F. Galiegue and K. Zyp. *JSON Schema: Core Definitions and Terminology*. Internet Engineering Task Force (IETF), 2013. URL https://tools.ietf.org/html/draft-zyp-json-schema-04.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979. ISSN 0098-3500. doi: 10.1145/355841.355847. URL http://doi.acm.org/10.1145/355841.355847.

Deborah Nolan and Duncan Temple Lang. *XML and Web Technologies for Data Sciences with R*. Springer-Verlag, 2014. URL http://link.springer.com/book/10.1007/978-1-4614-7900-0.

Jeroen Ooms, Duncan Temple Lang, and Jonathan Wallace. *jsonlite: A Smarter JSON Encoder for R*, 2014. URL http://github.com/jeroenooms/jsonlite#readme. R package version 0.9.8.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. URL http://www.R-project.org/.

Duncan Temple Lang. *RJSONIO: Serialize R Objects to JSON, JavaScript Object Notation*, 2013. URL http://CRAN.R-project.org/package=RJSONIO. R package version 1.0-3.

Hadley Wickham. Tidy Data. *Under review*, 2014. URL http://vita.had.co.nz/papers/tidy-data.pdf.