Package 'typedjson'

September 24, 2026

Title Type-Faithful and Human-Readable JSON for R Values

Version 0.1.1

Description Writing an R value as JSON that a human can read, and reading it back unchanged. The 'jsonlite' package offers either a readable but lossy pair of functions or a faithful but verbose one; this package emits ordinary JSON for ordinary values and annotates only what JSON cannot express, namely the distinction between integer and double, typed missing values, non-finite numbers, attributes, and objects from the S3, S4 and S7 systems.

License MIT + file LICENSE

Copyright The bundled 'yyjson' code in src/ is Copyright (c) 2020 YaoYuan and distributed under the MIT license. See file YYJSON-LICENSE for its full text.

Encoding UTF-8

Language en-US

RoxygenNote 8.0.0

Depends R (>= 4.3)

LinkingTo cpp11

Imports methods

Suggests jsonlite, knitr, R6, rmarkdown, S7, testthat (>= 3.0.0), withr

VignetteBuilder knitr, rmarkdown

Config/testthat/edition 3

URL https://nbenn.github.io/typedjson/

BugReports https://github.com/nbenn/typedjson/issues

NeedsCompilation yes

Author Nicolas Bennett [aut, cre, cph], YaoYuan [ctb, cph] (Author of the vendored yyjson library)

Maintainer Nicolas Bennett <nicolas@cynkra.com>

Repository CRAN

Date/Publication 2026-09-24 12:10:02 UTC

Contents

`json_read` *Write and read R values as JSON*

Description

Writing an R value produces JSON that a human can read, and reading it back produces the same value. Ordinary values are emitted as ordinary JSON; only what JSON cannot express is annotated, which keeps the document diffable, greppable and editable by hand.

Usage

```
json_read(path)

json_read_str(txt)

json_write(x, path, pretty = TRUE, typed = TRUE)

json_write_str(x, pretty = FALSE, typed = TRUE)
```

Arguments

`path`	Path to write to or read from.
`txt`	Document to read, as a length-one character vector.
`x`	Value to write.
`pretty`	Whether to indent the output. Files default to indented, because a persistence format is read in diffs; strings default to compact.
`typed`	Whether to record what JSON cannot express. The default writes the annotated form this package reads back unchanged; `FALSE` writes plain JSON for a consumer that brings its own schema.

Details

Two properties hold, and are what the test suite checks:

```
identical(json_read_str(json_write_str(x)), x)
json_write_str(json_read_str(doc)) == doc
```

The first holds for every supported value: every atomic type, missing values of each type, the non-finite doubles, attributes of any shape, language objects, closures, and objects built with S3, S4 or S7. Three values need it stated differently. An environment recorded by its contents comes back a new environment, which is the exception base R's own `serialize()` makes as well: what comes back binds the same names to the same values, locked the same way, under a parent that is itself equivalent, and a closure over one is equivalent for that same reason. An S7 class a document carries the definition of is equivalent for that same reason wherever a part of it closes over such an environment, which S7 builds for the constructor of every class that has a parent. A string R has not declared an encoding for comes back declared UTF-8, so the property holds on its bytes rather than under `identical()`. The second holds for every document this package can write. Foreign documents are read under the same grammar and normalize on the first round trip, since a mixed-type array such as `[1, "a"]` has to come back as a list. Two things are refused instead of normalized, both inside the namespace the `~` prefix reserves. A key beginning with a single `~` is a format tag, and one this reader does not know is an error rather than a name. A string beginning with `~` and a reserved discriminator, which is `z` for what JSON has no lexeme for and `:` for a symbol, is a tag as well, and an unknown one is an error rather than text; every other tilde-leading string stays a string, so `~/data` is a path.

Two rules decide the shape of a document. A JSON array of scalars is an atomic vector and a JSON object is a named list, so the two containers mean what they mean everywhere else. A length-one vector is written bare wherever an array could not be mistaken for it, which is at the document root, as an object value, as an attribute and as the payload of a tagged object; only as an array element does it keep its brackets, because there the brackets are the sole thing separating `list(1, 2)` from `c(1, 2)`.

Some JSON is written for a consumer that already defines the shape it expects. There R's types are noise, since the document has to satisfy an external schema rather than describe the value it came from, and the `typed` flag says so. Plain mode is this format with the annotations left out: the two container rules and the number lexemes stay, the S4 bit goes, and a length-one vector renders as a scalar unless it is `AsIs`, in which case it keeps its brackets. That last rule is not a setting, because shape requirements run both ways inside one document — a schema wants a scalar at `additionalProperties` and an array at `required` whatever its length — and no encoder can tell the two apart by looking, since at length one a scalar and a one-element array are the same R object. The distinction already lives in the value, where `I("x")` differs from `"x"`, so plain mode renders it rather than importing a policy for it.

Attributes are not quite dropped wholesale, and the rule that decides which two survive is worth stating, because it predicts the rest. JSON puts two questions to every value that the container rules leave open — object or array, and at length one scalar or array — and plain mode reads the attributes that answer them: the `names` of a list, and the `AsIs` marker. Nothing else is asked anything. A `dim` answers neither, since a vector is a flat array with one or without one, so a matrix flattens; the names of an atomic vector answer neither either, since an atomic vector is an array whatever its elements are called; and `levels`, `tzone`, `units` and a class naming a type carry meaning rather than shape, so a factor writes its codes and a `Date` its number.

Two consequences are worth naming, because in both plain mode writes what the default refuses. Nothing walks into an attribute, so a handle that is only reachable through one is dropped along with it: a connection is an integer wearing an external pointer, and where the default stops at that pointer, plain mode never reaches it and writes the bare slot index. And a `json_state()` method is not consulted, since a method says how to persist a value and plain mode is not persistence — so a method written to keep a field out of a document does not stand between `typed = FALSE` and that

field. Both follow from the rule above rather than qualifying it, and `vignette("handles")` argues the default's side of each.

Nothing is written in plain mode that the value is not. A missing value becomes `null`, which is what JSON spells absence with, and everything the annotations were the only way to write is refused where it sits, naming the path: complex and raw values, the non-finite doubles, symbols, calls, closures and environments. What plain mode does not do is escape, since the consumer asked for the name and the string it asked for, so a value carrying a leading `~` of its own reaches the document bare and is read back the way any foreign document carrying one is: a key spelling a tag this reader does not know is refused, and a string spelling one it does know comes back as that tag. Reading a plain document returns what the document says rather than the value that wrote it, which is what the default mode is for.

Text is carried as UTF-8. A string R has declared as UTF-8 or latin1 is converted from what it declares, and one it has not declared is taken as the bytes it holds rather than translated through the locale, so the same value writes the same document on every machine. Undeclared bytes that are not valid UTF-8 have no reading to fall back on and are refused, naming the path they sit at, as is a string declared with the `"bytes"` encoding. A document carries one encoding, so every string read out of one comes back marked UTF-8, and an undeclared string therefore revives declared. Comparing the two with `identical()` translates the native one through the locale and so disagrees wherever that locale cannot represent the bytes; the bytes themselves are the same in every locale, and only the declaration has moved.

An environment is recorded by name wherever a name finds it again: the global, base and empty environments, a namespace, a package environment and the imports environment of a namespace. Those come back as the object they were written from. Anything else is recorded by its contents, with the parent following the same rule and the locked bit and locked bindings recorded alongside, and comes back equivalent. A recorded name that is not available on the way back is replaced by the global environment with a warning, the way base R already does.

Reference identity is recorded across a document. A reference the walk reaches more than once is numbered with a `~id` where it is first written, and each later position carries a `~ref` naming that number rather than a second copy, so positions holding one environment on the way in hold one environment on the way back. Nothing is numbered where nothing repeats, which leaves a document carrying no sharing as it was. A cycle rides the same numbering, since an environment is built and numbered before what it binds is read: an environment whose parent frame binds it back comes back bound that way. What stays refused, naming both ends of it, is a cycle closing through an object the extension protocol builds in one call, an opted-in `R6` instance among them, since a constructor cannot be handed an object that already exists.

A language object is a value rather than a handle, so it round-trips exactly and nothing about it is deparsed. A call, an expression and a pairlist are written as their elements under a `~t` naming the type, keeping the argument names R stores as tags, and a symbol is written as the string `~:name`. Attributes ride the ordinary rule, which is what carries the `.Environment` of a formula, so `y ~ x` round-trips once an environment does.

A closure compares by its parts rather than by reference, so it is a value as well and round-trips as far as its environment does. Formals, body and environment are written under a `~t` of `closure`, and attributes ride the ordinary rule. A primitive closes over nothing and is recorded by the name that finds it again, which is what base R does with one. No source reference is recorded: the `srcref` a parser attaches to a function definition, to a `{` block and to what `parse()` returns is dropped, which keeps a document diffable and leaves the value equal under `identical()`, whose default ignores

one. A byte-compiled closure is written from `body()`, which is the source tree it was compiled from.

Values that are handles rather than data stay out: an external pointer is refused rather than silently written as something else, and an environment binding holding one is refused with it. So is a binding holding a promise, since forcing it on the writer's own initiative could run arbitrary code, and an active binding, since reading it would do the same and record the result as though it were a plain value. That reaches a closure through the frame it closes over, where an argument the function was called with stays a promise whether or not it has been forced. A class that owns such a handle can still be persisted by writing a `json_state()` method for it.

Slots and properties are attributes, so an S4 or S7 object is rebuilt by the attribute rule rather than by whatever the class constructs one with. The check the class does supply is run on the way back — `methods::validObject()` for S4 and `S7::validate()` for S7 — so a document edited into a value the class rejects is refused where it is read, and an S7 property the document leaves out or spells as the wrong type is caught with it. What the check does not stand in for is the construction it reached past: an `initialize` method for S4 and a custom `constructor` for S7 do not run, so a slot or property one of them would have derived comes back as the document spells it. A class this session does not hold leaves nothing to check against, and a document naming one reads as it always has.

An S7 class is recorded by the name that finds it again wherever one does, and by its definition wherever none does. S7 sets `package` for a class defined in a package and leaves it `NULL` for every class defined outside one, which is the same question, so that attribute is what decides: a package-scoped class is recorded as the class and package it names, and a class with no package carries its parent, properties, constructor and validator instead, so a document written from it reads in a session where no such class exists. Every class those parts name is recorded by what identifies it rather than walked into — one S7 itself binds by the name it holds there, an S3 class by its class vector, a union by its members, and a class of your own by the two forms above — since every refusal a walk into one hits is inside S7's own machinery rather than in the class you wrote. The class vector an object records is checked against the class it resolves to, so a document where the two disagree is refused rather than dispatching on the one and taking its properties from the other.

An `R6` instance is refused as well, for a reason one level up. What an `R6` class guarantees is what its methods say rather than what its bindings happen to hold, so those bindings are not a value the package can record on the class's behalf, and the refusal names the class and the method it wants. A class author who has decided the bindings are the state opts in with that method, and `r6_state()` is the pair recording them. An `R6` class generator needs no method either way: it is recorded by the class it names, and comes back the object it was written from.

A reference class instance is refused on the second half of that reason alone. Fields are declared there, so what the representation is already has an answer, but the object is still a reference whose `initialize` may establish an invariant and whose fields hold whatever a private binding would, so a method of your own is what settles it here as well, on the concrete class or on any class between that and `envRefClass`, which is where the refusal sits. The generator that makes one is refused rather than recorded, since a walk into it reaches the internals of the `methods` package rather than the class. An environment you have classed yourself claims none of this, and is written by the environment rule above, contents and all.

Value

The `json_write()` function returns `path` invisibly and `json_write_str()` a length-one character vector. Both readers return the value the document describes.

Examples

```
json_write_str(list(n = 1L, x = 2.5, missing = NA_character_))

json_write_str(as.Date("2026-01-01"))

json_write_str(quote(mpg ~ wt))

json_write_str(stats::median)

x <- c(a = 1, b = Inf)
identical(json_read_str(json_write_str(x)), x)

json_write_str(list(required = I("x"), additionalProperties = FALSE),
               typed = FALSE)
```

`json_state` *Persist a class the default rule does not fit*

Description

Most classes need nothing here: an S3, S4 or S7 object is a base type plus attributes, so `json_write()` records it without help. A class whose instances hold something outside that model — a connection opened by `initialize`, a handle to a running process, a reference that has to be recorded as a key rather than a value — supplies a method for this pair instead, modeled on the `__getstate__` and `__setstate__` protocol of Python's pickle.

Usage

```
json_state(x)

json_revive(class, state)
```

Arguments

`x`	Object whose state is to be recorded.
`class`	Empty object carrying the recorded class vector, which `json_revive()` dispatches on.
`state`	Whatever the matching `json_state()` method returned.

Details

A `json_state()` method returns a plain list of what to persist, and is free to leave out anything that can be recomputed. The document records that list next to the classes the object dispatches on, which for an S4 object or a reference class instance is its inheritance chain rather than the concrete class alone. On the way back, `json_revive()` dispatches on the recorded classes through an empty object carrying them, so a method signature always starts with the class token rather than the object being rebuilt, and a method registered on a superclass is reached both ways.

Methods for the class generators of both `R6` and S7 ship with the package and follow the same protocol. An `R6` instance has no method, and writing one is refused rather than guessed at; see `r6_state()` for why, and for the pair a class author opts in with. A reference class instance is refused on the same grounds. A method on the concrete class settles it, as does one on any class between that and `envRefClass`, which is where the refusal itself sits. The generator that makes one is refused outright, since a walk into it reaches the internals of the `methods` package rather than the class.

Value

The `json_state()` function returns a list, and `json_revive()` the rebuilt object.

Examples

```
handle <- structure(list(path = "/tmp/log", con = "a live connection"),
                    class = "file_handle")

json_state.file_handle <- function(x) list(path = x$path)

json_revive.file_handle <- function(class, state) {
  structure(list(path = state$path, con = NULL), class = "file_handle")
}

json_write_str(handle)

json_read_str(json_write_str(handle))
```

`r6_state` *Record an* `R6` *instance as the bindings it holds*

Description

An `R6` instance has no `json_state()` method of its own, because what an `R6` class guarantees is what its methods say rather than what its bindings happen to hold: a private field is private precisely because it is not part of that contract. Writing one is therefore refused, naming the class and the method it wants.

Usage

```
r6_state(x)

r6_restore(class, state)
```

Arguments

x	The R6 instance whose bindings are to be recorded.
class	Empty object carrying the recorded class vector, which json_revive() dispatches on.
state	Whatever r6_state() returned.

Details

A class author is the party who knows whether a field is stored or derived, whether initialize establishes an invariant, and whether a reference should be recorded as a key rather than a value. One who has made that judgment and wants the instance recorded as its bindings anyway opts in with one method each way:

```
json_state.MyClass <- function(x) r6_state(x)
json_revive.MyClass <- function(class, state) r6_restore(class, state)
```

The pair records the class's package alongside the public and private bindings an instance holds, leaving out anything the generator chain declares as a method and anything bound actively. On the way back the generator is found again by name, an instance is allocated from a twin of it whose initialize does nothing, and the recorded bindings are written into that.

Value

The r6_state() function returns a list carrying package, public and private, and r6_restore() the rebuilt instance.

What the pair does not guarantee

The mechanism reaches past the interface the class offers, which is what makes it the author's call rather than the default. Four consequences are worth stating outright.

Bindings are reinstated past initialize. The instance that comes back is filled from the document rather than constructed, so an invariant initialize establishes is not re-established and a resource it acquires is not acquired. A finalize method still registers on the object, and so runs against a handle it never held.

The round trip is exact only while the class's shape is unchanged. Where the generator locks its instances, recorded state the class no longer declares has nowhere to go and is dropped with a warning naming it; a field the class has gained since arrives at its default. The lock itself comes from the generator, so one placed on a single object or binding by hand is not recorded and does not come back.

A revived instance is a new environment, so the trip holds up to the equivalence vignette("design") states for an environment recorded by its contents rather than under identical().

Both halves need the generator to be findable by name in the environment the class was defined in, which is checked on the way out as well as on the way in. A class defined inside a function, one whose name finds two generators, and a non-portable class are all refused where they are written.

Examples

```
Counter <- R6::R6Class("Counter",
  public = list(
    n = 0,
    initialize = function(n = 0) self$n <- n,
    bump = function() {
      self$n <- self$n + 1
      invisible(self)
    }
  )
)

json_state.Counter <- function(x) r6_state(x)
json_revive.Counter <- function(class, state) r6_restore(class, state)

counter <- Counter$new()$bump()$bump()

json_read_str(json_write_str(counter))$n
```

Index