Package: jsonvalidate (via r-universe)

September 21, 2024

Title Validate 'JSON' Schema

Version 1.4.2

Maintainer Rich FitzJohn <rich.fitzjohn@gmail.com>

Description Uses the node library 'is-my-json-valid' or 'ajv' to validate 'JSON' against a 'JSON' schema. Drafts 04, 06 and 07 of 'JSON' schema are supported.

License MIT + file LICENSE

URL https://docs.ropensci.org/jsonvalidate/, https://github.com/ropensci/jsonvalidate

BugReports https://github.com/ropensci/jsonvalidate/issues

Imports R6, V8

Suggests knitr, jsonlite, rmarkdown, testthat, withr

RoxygenNote 7.2.3

Roxygen list(markdown = TRUE)

VignetteBuilder knitr

Encoding UTF-8

Language en-GB

Config/testthat/edition 3

Repository https://vimc.r-universe.dev

RemoteUrl https://github.com/ropensci/jsonvalidate

RemoteRef master

RemoteSha f942ba39f09474ffee82fef35761b6e1cde0165f

Contents

`json_schema` *Interact with JSON schemas*

Description

Interact with JSON schemas, using them to validate json strings or serialise objects to JSON safely.

This interface supercedes json_schema and changes some default arguments. While the old interface is not going away any time soon, users are encouraged to switch to this interface, which is what we will develop in the future.

Public fields

`schema` The parsed schema, cannot be rebound

`engine` The name of the schema validation engine

Methods

Public methods:

- `json_schema$new()`
- `json_schema$validate()`
- `json_schema$serialise()`

Method `new()`**:** Create a new `json_schema` object.

Usage:

```
json_schema$new(schema, engine = "ajv", reference = NULL, strict = FALSE)
```

Arguments:

`schema` Contents of the json schema, or a filename containing a schema.

`engine` Specify the validation engine to use. Options are "ajv" (the default; "Another JSON Schema Validator") or "imjv" ("is-my-json-valid", the default everywhere in versions prior to 1.4.0, and the default for json_validator. *Use of* `ajv` *is strongly recommended for all new code*.

`reference` Reference within schema to use for validating against a sub-schema instead of the full schema passed in. For example if the schema has a 'definitions' list including a definition for a 'Hello' object, one could pass "#/definitions/Hello" and the validator would check that the json is a valid "Hello" object. Only available if `engine = "ajv"`.

`strict` Set whether the schema should be parsed strictly or not. If in strict mode schemas will error to "prevent any unexpected behaviours or silently ignored mistakes in user schema". For example it will error if encounters unknown formats or unknown keywords. See https://ajv.js.org/strict-mode.html for details. Only available in `engine = "ajv"` and silently ignored for "imjv". Validate a json string against a schema.

Method `validate()`**:**

Usage:

```
json_schema$validate(
  json,
  verbose = FALSE,
  greedy = FALSE,
  error = FALSE,
  query = NULL
)
```

Arguments:

`json` Contents of a json object, or a filename containing one.

`verbose` Be verbose? If `TRUE`, then an attribute "errors" will list validation failures as a data.frame

`greedy` Continue after the first error?

`error` Throw an error on parse failure? If `TRUE`, then the function returns `NULL` on success (i.e., call only for the side-effect of an error on failure, like `stopifnot`).

`query` A string indicating a component of the data to validate the schema against. Eventually this may support full jsonpath syntax, but for now this must be the name of an element within `json`. See the examples for more details. Serialise an R object to JSON with unboxing guided by the schema. See json_serialise for details on the problem and the algorithm.

Method `serialise()`**:**

Usage:

```
json_schema$serialise(object)
```

Arguments:

`object` An R object to serialise

Examples

```
# This is the schema from ?json_validator
schema <- '{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Product",
    "description": "A product from Acme\'s catalog",
    "type": "object",
    "properties": {
        "id": {
            "description": "The unique identifier for a product",
            "type": "integer"
        },
        "name": {
            "description": "Name of the product",
            "type": "string"
        },
        "price": {
            "type": "number",
            "minimum": 0,
            "exclusiveMinimum": true
        },
        "tags": {
            "type": "array",
            "items": {
```

```
                "type": "string"
            },
            "minItems": 1,
            "uniqueItems": true
        }
    },
    "required": ["id", "name", "price"]
}'

# We're going to use a validator object below
v <- jsonvalidate::json_validator(schema, "ajv")

# And this is some data that we might generate in R that we want to
# serialise using that schema
x <- list(id = 1, name = "apple", price = 0.50, tags = "fruit")

# If we serialise to json, then 'id', 'name' and "price' end up a
# length 1-arrays
jsonlite::toJSON(x)

# ...and that fails validation
v(jsonlite::toJSON(x))

# If we auto-unbox then 'fruit' ends up as a string and not an array,
# also failing validation:
jsonlite::toJSON(x, auto_unbox = TRUE)
v(jsonlite::toJSON(x, auto_unbox = TRUE))

# Using json_serialise we can guide the serialisation process using
# the schema:
jsonvalidate::json_serialise(x, schema)

# ...and this way we do pass validation:
v(jsonvalidate::json_serialise(x, schema))

# It is typically much more efficient to construct a json_schema
# object first and do both operations with it:
obj <- jsonvalidate::json_schema$new(schema)
json <- obj$serialise(x)
obj$validate(json)
```

`json_serialise` *Safe JSON serialisation*

Description

Safe serialisation of json with unboxing guided by the schema.

Usage

```
json_serialise(
  object,
  schema,
  engine = "ajv",
  reference = NULL,
  strict = FALSE
)
```

Arguments

`object`	An object to be serialised
`schema`	A schema (string or path to a string, suitable to be passed through to json_validator or a validator object itself.
`engine`	The engine to use. Only ajv is supported, and trying to use `imjv` will throw an error.
`reference`	Reference within schema to use for validating against a sub-schema instead of the full schema passed in. For example if the schema has a 'definitions' list including a definition for a 'Hello' object, one could pass "#/definitions/Hello" and the validator would check that the json is a valid "Hello" object. Only available if `engine = "ajv"`.
`strict`	Set whether the schema should be parsed strictly or not. If in strict mode schemas will error to "prevent any unexpected behaviours or silently ignored mistakes in user schema". For example it will error if encounters unknown formats or unknown keywords. See https://ajv.js.org/strict-mode.html for details. Only available in `engine = "ajv"`.

Details

When using jsonlite::toJSON we are forced to deal with the differences between R's types and those available in JSON. In particular:

- R has no scalar types so it is not clear if `1` should be serialised as a number or a vector of length 1; jsonlite provides support for "automatically unboxing" such values (assuming that length-1 vectors are scalars) or never unboxing them unless asked to using jsonlite::unbox
- JSON has no date/time values and there are many possible string representations.
- JSON has no data.frame or matrix type and there are several ways of representing these in JSON, all equally valid (e.g., row-wise, column-wise or as an array of objects).
- The handling of `NULL` and missing values (`NA`, `NaN`) are different
- We need to chose the number of digits to write numbers out at, balancing precision and storage.

These issues are somewhat lessened when we have a schema because we know what our target type looks like. This function attempts to use the schema to guide serialsation of json safely. Currently it only supports detecting the appropriate treatment of length-1 vectors, but we will expand functionality over time.

For a user, this function provides an argument-free replacement for `jsonlite::toJSON`, accepting an R object and returning a string with the JSON representation of the object. Internally the algorithm is:

1. serialise the object with jsonlite::toJSON, with `auto_unbox = FALSE` so that length-1 vectors are serialised as a length-1 arrays.
2. operating entirely within JavaScript, deserialise the object with `JSON.parse`, traverse the object and its schema simultaneously looking for length-1 arrays where the schema says there should be scalar value and unboxing these, and re-serialise with `JSON.stringify`

There are several limitations to our current approach, and not all unboxable values will be found - at the moment we know that schemas contained within a `oneOf` block (or similar) will not be recursed into.

Value

A string, representing `object` in JSON format. As for `jsonlite::toJSON` we set the class attribute to be `json` to mark it as serialised json.

Warning:

Direct use of this function will be slow! If you are going to serialise more than one or two objects with a single schema, you should use the `serialise` method of a json_schema object which you create once and pass around.

Examples

```
# This is the schema from ?json_validator
schema <- '{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Product",
    "description": "A product from Acme\'s catalog",
    "type": "object",
    "properties": {
        "id": {
            "description": "The unique identifier for a product",
            "type": "integer"
        },
        "name": {
            "description": "Name of the product",
            "type": "string"
        },
        "price": {
            "type": "number",
            "minimum": 0,
            "exclusiveMinimum": true
        },
        "tags": {
            "type": "array",
            "items": {
                "type": "string"
            },
            "minItems": 1,
            "uniqueItems": true
        }
    },
```

```
    "required": ["id", "name", "price"]
}'

# We're going to use a validator object below
v <- jsonvalidate::json_validator(schema, "ajv")

# And this is some data that we might generate in R that we want to
# serialise using that schema
x <- list(id = 1, name = "apple", price = 0.50, tags = "fruit")

# If we serialise to json, then 'id', 'name' and "price' end up a
# length 1-arrays
jsonlite::toJSON(x)

# ...and that fails validation
v(jsonlite::toJSON(x))

# If we auto-unbox then 'fruit' ends up as a string and not an array,
# also failing validation:
jsonlite::toJSON(x, auto_unbox = TRUE)
v(jsonlite::toJSON(x, auto_unbox = TRUE))

# Using json_serialise we can guide the serialisation process using
# the schema:
jsonvalidate::json_serialise(x, schema)

# ...and this way we do pass validation:
v(jsonvalidate::json_serialise(x, schema))

# It is typically much more efficient to construct a json_schema
# object first and do both operations with it:
obj <- jsonvalidate::json_schema$new(schema)
json <- obj$serialise(x)
obj$validate(json)
```

`json_validate` *Validate a json file*

Description

Validate a single json against a schema. This is a convenience wrapper around `json_validator(schema)(json)` or `json_schema$new(schema, engine = "ajv")$validate(json)`. See `json_validator()` for further details.

Usage

```
json_validate(
  json,
  schema,
  verbose = FALSE,
```

```
  greedy = FALSE,
  error = FALSE,
  engine = "imjv",
  reference = NULL,
  query = NULL,
  strict = FALSE
)
```

Arguments

json	Contents of a json object, or a filename containing one.
schema	Contents of the json schema, or a filename containing a schema.
verbose	Be verbose? If TRUE, then an attribute "errors" will list validation failures as a data.frame
greedy	Continue after the first error?
error	Throw an error on parse failure? If TRUE, then the function returns NULL on success (i.e., call only for the side-effect of an error on failure, like stopifnot).
engine	Specify the validation engine to use. Options are "imjv" (the default; which uses "is-my-json-valid") and "ajv" (Another JSON Schema Validator). The latter supports more recent json schema features.
reference	Reference within schema to use for validating against a sub-schema instead of the full schema passed in. For example if the schema has a 'definitions' list including a definition for a 'Hello' object, one could pass "#/definitions/Hello" and the validator would check that the json is a valid "Hello" object. Only available if engine = "ajv".
query	A string indicating a component of the data to validate the schema against. Eventually this may support full jsonpath syntax, but for now this must be the name of an element within json. See the examples for more details.
strict	Set whether the schema should be parsed strictly or not. If in strict mode schemas will error to "prevent any unexpected behaviours or silently ignored mistakes in user schema". For example it will error if encounters unknown formats or unknown keywords. See https://ajv.js.org/strict-mode.html for details. Only available in engine = "ajv".

Examples

```
# A simple schema example:
schema <- '{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Product",
    "description": "A product from Acme\'s catalog",
    "type": "object",
    "properties": {
        "id": {
            "description": "The unique identifier for a product",
            "type": "integer"
        },
```

```
        "name": {
            "description": "Name of the product",
            "type": "string"
        },
        "price": {
            "type": "number",
            "minimum": 0,
            "exclusiveMinimum": true
        },
        "tags": {
            "type": "array",
            "items": {
                "type": "string"
            },
            "minItems": 1,
            "uniqueItems": true
        }
    },
    "required": ["id", "name", "price"]
}'

# Test if some (invalid) json conforms to the schema
jsonvalidate::json_validate("{}", schema, verbose = TRUE)

# Test if some (valid) json conforms to the schema
json <- '{
    "id": 1,
    "name": "A green door",
    "price": 12.50,
    "tags": ["home", "green"]
}'
jsonvalidate::json_validate(json, schema)

# Test a fraction of a data against a reference into the schema:
jsonvalidate::json_validate(json, schema,
                            query = "tags", reference = "#/properties/tags",
                            engine = "ajv", verbose = TRUE)
```

`json_validator` *Create a json validator*

Description

Create a validator that can validate multiple json files.

Usage

```
json_validator(schema, engine = "imjv", reference = NULL, strict = FALSE)
```

Arguments

schema	Contents of the json schema, or a filename containing a schema.
engine	Specify the validation engine to use. Options are "imjv" (the default; which uses "is-my-json-valid") and "ajv" (Another JSON Schema Validator). The latter supports more recent json schema features.
reference	Reference within schema to use for validating against a sub-schema instead of the full schema passed in. For example if the schema has a 'definitions' list including a definition for a 'Hello' object, one could pass "#/definitions/Hello" and the validator would check that the json is a valid "Hello" object. Only available if `engine = "ajv"`.
strict	Set whether the schema should be parsed strictly or not. If in strict mode schemas will error to "prevent any unexpected behaviours or silently ignored mistakes in user schema". For example it will error if encounters unknown formats or unknown keywords. See https://ajv.js.org/strict-mode.html for details. Only available in `engine = "ajv"`.

Value

A function that can be used to validate a schema. Additionally, the function has two attributes assigned: `v8` which is the javascript context (used internally) and `engine`, which contains the name of the engine used.

Validation Engines

We support two different json validation engines, `imjv` ("is-my-json-valid") and `ajv` ("Another JSON Validator"). `imjv` was the original validator included in the package and remains the default for reasons of backward compatibility. However, users are encouraged to migrate to `ajv` as with it we support many more features, including nested schemas that span multiple files, meta schema versions later than draft-04, validating using a subschema, and validating a subset of an input data object.

If your schema uses these features we will print a message to screen indicating that you should update when running interactively. We do not use a warning here as this will be disruptive to users. You can disable the message by setting the option `jsonvalidate.no_note_imjv` to `TRUE`. Consider using `withr::with_options()` (or simply `suppressMessages()`) to scope this option if you want to quieten it within code you do not control. Alternatively, setting the option `jsonvalidate.no_note_imjv` to `FALSE` will print the message even noninteractively.

Updating the engine should be simply a case of adding `engine = "ajv"` to your `json_validator` or `json_validate` calls, but you may see some issues when doing so.

- Your json now fails validation: We've seen this where schemas spanned several files and are silently ignored. By including these, your data may now fail validation and you will need to either fix the data or the schema.
- Your code depended on the exact payload returned by `imjv`: If you are inspecting the error result and checking numbers of errors, or even the columns used to describe the errors, you will likely need to update your code to accommodate the slightly different format of `ajv`
- Your schema is simply invalid: If you reference an invalid metaschema for example, jsonvalidate will fail

Using multiple files

Multiple files are supported. You can have a schema that references a file `child.json` using `{"$ref": "child.json"}`—in this case if `child.json` includes an `id` or `$id` element it will be silently dropped and the filename used to reference the schema will be used as the schema id.

The support is currently quite limited - it will not (yet) read sub-child schemas relative to child schema `$id` url, and does not support reading from URLs (only local files are supported).

Examples

```
# A simple schema example:
schema <- '{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "Product",
    "description": "A product from Acme\'s catalog",
    "type": "object",
    "properties": {
        "id": {
            "description": "The unique identifier for a product",
            "type": "integer"
        },
        "name": {
            "description": "Name of the product",
            "type": "string"
        },
        "price": {
            "type": "number",
            "minimum": 0,
            "exclusiveMinimum": true
        },
        "tags": {
            "type": "array",
            "items": {
                "type": "string"
            },
            "minItems": 1,
            "uniqueItems": true
        }
    },
    "required": ["id", "name", "price"]
}'

# Create a validator function
v <- jsonvalidate::json_validator(schema)

# Test if some (invalid) json conforms to the schema
v("{}", verbose = TRUE)

# Test if some (valid) json conforms to the schema
v('{
    "id": 1,
    "name": "A green door",
```

```
    "price": 12.50,
    "tags": ["home", "green"]
}')

# Using features from draft-06 or draft-07 requires the ajv engine:
schema <- "{
  '$schema': 'http://json-schema.org/draft-06/schema#',
  'type': 'object',
  'properties': {
    'a': {
      'const': 'foo'
    }
  }
}"

# Create the validator
v <- jsonvalidate::json_validator(schema, engine = "ajv")

# This confirms to the schema
v('{"a": "foo"}')

# But this does not
v('{"a": "bar"}')
```

Index