schema-inspector

2.0.2 • Public • Published a month ago

Readme

Code Beta

1 Dependency

111 Dependents

43 Versions

schema-inspector

Schema-Inspector is a powerful tool to sanitize and validate JS objects. It's designed to work both client-side and server-side and to be scalable with allowing asynchronous and synchronous calls.

See a live example: http://schema-inspector.github.io/schema-inspector/

Installation

Node.js

```
npm install schema-inspector
```

Browser

Bower uses have reported success using the library this way, using bower overrides in `bower.json`.

```
93  "overrides": {
94    "schema-inspector": {
95      "main": "lib/schema-inspector.js"
96    },
```

Bower is not officially-supported as a build tool and references to it will be removed from the repository in versions 3.0.0+.

Comparison with JSON Schema

`schema-inspector` is not compatable with JSON Schema. They are two different ways to validate data. However, the main difference is that `schema-inspector` supports sanitization of data.

Version 2.0.0

To fix a security vulnerability in the 1.x.x email Regex expression used, a new Regex expression was used which may be less flexible than the expression used in 1.x.x. Therefore, version 2.0.0 was released with this new expression. It's highly-recommended to upgrade to this new version after testing it.

If you need the old, insecure behavior, use version 1.x.x or use the custom validation function feature for your field and perform email address validation any way you like.

How it looks like

Click to see it live!

Usage

```
var inspector = require('schema-inspector');

// Data that we want to sanitize and validate
var data = {
    firstname: 'sterling  ',
    lastname: '  archer',
    jobs: 'Special agent, cocaine Dealer',
    email: 'NEVER!',
};

// Sanitization Schema
var sanitization = {
    type: 'object',
    properties: {
        firstname: { type: 'string', rules: ['trim', 'title'] },
        lastname: { type: 'string', rules: ['trim', 'title'] },
        jobs: {
            type: 'array',
            splitWith: ',',
            items: { type: 'string', rules: ['trim', 'title'] },
        },
        email: { type: 'string', rules: ['trim', 'lower'] },
    },
};
// Let's update the data
inspector.sanitize(sanitization, data);
/*
data is now:
{
    firstname: 'Sterling',
    lastname: 'Archer',
    jobs: ['Special Agent', 'Cocaine Dealer'],
    email: 'never!'
}
*/

// Validation schema
var validation = {
    type: 'object',
    properties: {
        firstname: { type: 'string', minLength: 1 },
        lastname: { type: 'string', minLength: 1 },
        jobs: {
            type: 'array',
            items: { type: 'string', minLength: 1 },
        },
        email: { type: 'string', pattern: 'email' },
    },
};
```

```
var result = inspector.validate(validation, data);
if (!result.valid)
    console.log(result.format());
/*
 Property @.email: must match [email], but is equal to "never!"
*/
```

Tips: it's recommended to use one schema for the sanitization and another for the validation,

In the browser

```
<script type="text/javascript" src="async.js"></script>
<script type="text/javascript" src="schema-inspector.js"></script>
<script type="text/javascript">
    var schema = { /* ... */ };
    var candidate = { /* ... */ };
    SchemaInspector.validate(schema, candidate, function (err, result) {
    if (!result.valid)
        return alert(result.format());
 });
</script>
```

In the example below, the `inspector` variable will be used. For the client-side use `SchemaInspector` instead of `inspector`.

Documentation

Validation

- type
- optional
- pattern
- minLength, maxLength, exactLength
- lt, lte, gt, gte, eq, ne
- someKeys
- strict
- exec
- properties
- items
- alias
- error
- code

Sanitization

- type
- def
- optional
- rules
- min, max
- minLength, maxLength
- strict
- exec
- properties
- items

Custom fields

- punctual use
- extension
- context

Asynchronous call

- How to

Thanks to

- Benjamin Gressier (major contributor of this awesome module)

Validation

type

- **type**: string, array of string.
- **usable on**: any.
- **possible values**
  - `string`
  - `number`
  - `integer`
  - `boolean`
  - `null`
  - `date` (instanceof Date), you can use the `validDate: true` to check if the date is valid
  - `object` (typeof element === 'object') *Note: array, null, or dates don't match the object type*
  - `array` (constructor === Array)
  - A function (candidate isinstance)
  - `any` (it can be anything)

Allow to check property type. If the given value is incorrect, then type is not checked.

Example

```
var inspector = require('schema-inspector');

function Class() {}

var schema = {
    type: 'object',
    properties: {
        lorem: { type: 'number' },
        ipsum: { type: 'any' },
        dolor: { type: ['number', 'string', 'null'] },
        sit: { type: Class },
    },
};

var c1 = {
    lorem: 12,
    ipsum: 'sit amet',
    dolor: 23,
    sit: new Class(),
};
var c2 = {
    lorem: 12,
    ipsum: 34,
    dolor: 'sit amet',
    sit: new Class(),
};
var c3 = {
    lorem: 12,
    ipsum: ['sit amet'],
    dolor: null,
    sit: new Class(),
};
var c4 = {
    lorem: '12',
    ipsum: 'sit amet',
    dolor: new Date(),
    sit: {},
};

inspector.validate(schema, c1); // Valid
inspector.validate(schema, c2); // Valid
inspector.validate(schema, c3); // Valid
inspector.validate(schema, c4); // Invalid: @.lorem must be a number, @dolor must be a number, a string or null, @.sit must be an ins
```

optional

- **type**: boolean.

- **default**: false.
- **usable on**: any.

This field indicates whether or not property has to exist.

Example

```
var inspector = require('Roadspector');

var schema1 = {
    type: 'object',
    properties: {
        lorem: { type: 'any', optional: true },
    },
};

var schema2 = {
    type: 'object',
    properties: {
        lorem: { type: 'any', optional: false }, // default value
    },
};

var c1 = { lorem: 'ipsum' };
var c2 = {};

inspector.validate(schema1, c1); // Valid
inspector.validate(schema1, c2); // Valid
inspector.validate(schema2, c1); // Valid
inspector.validate(schema2, c2); // Invalid: "@.lorem" is missing and not optional
```

---

uniqueness

- **type**: boolean.
- **default**: false.
- **usable on**: array, string.

If true, then we ensure no element in candidate exists more than once.

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'array',
    uniqueness: true,
};

var c1 = [12, 23, 34, 45];
var c2 = [12, 23, 34, 12];

inspector.validate(schema, c1); // Valid
inspector.validate(schema, c2); // Invalid: 12 exists twice in @.
```

---

pattern

- **type**: string, RegExp object, array of string and RegExp.
- **usable on**: string.
- Possible values as a string: `void`, `url`, `date-time`, `date`, `coolDateTime`, `time`, `color`, `email`, `numeric`, `integer`, `decimal`, `v4uuid`, `alpha`, `alphaNumeric`, `alphaDash`, `javascript`, `upperString`, `lowerString`.

Ask Schema-Inspector to check whether or not a given matches provided patterns. When a pattern is a RegExp, it directly test the string with it. When it's a string, it's an alias of a RegExp.

Example

```
var inspector = require('schema-inspector');

var schema1 = {
    type: 'array',
    items: { type: 'string', pattern: /^[A-C]/ },
};
```

```
var c1 = ['Alorem', 'Bipsum', 'Cdolor', 'DSit amet'];

var schema2 = {
    type: 'array',
    items: { type: 'string', pattern: 'email' },
};

var c2 = ['lorem@ipsum.com', 'dolor@sit.com', 'amet@consectetur'];

inspector.validate(schema1, c1); // Invalid: @[3] ('DSit amet') does not match /^[A-C]/
inspector.validate(schema2, c2); // Invalid: @[2] ('amet@consectetur') does not match "email" pattern.
```

minLength, maxLength, exactLength

- **type**: integer.
- **usable on**: array, string.

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    properties: {
        lorem: { type: 'string', minLength: 4, maxLength: 8 },
        ipsum: { type: 'array', exactLength: 6 },
    },
};
var c1 = {
    lorem: '12345',
    ipsum: [1, 2, 3, 4, 5, 6],
};

var c2 = {
    lorem: '123456789',
    ipsum: [1, 2, 3, 4, 5],
};

inspector.validate(schema, c1); // Valid
inspector.validate(schema, c2); // Invalid: @.lorem must have a length between 4 and 8 (here 9)
// and @.ipsum must have a length of 6 (here 5)
```

lt, lte, gt, gte, eq, ne

- **type**: number (string, number and boolean for eq).
- **usable on**: number (string, number and boolean for eq).

Check whether comparison is true:

- lt: <
- lte: <=
- gt: >
- gte: >=
- eq: ===
- ne: !==

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    properties: {
        lorem: { type: 'number', gt: 0, lt: 5 }, // Between ]0; 5[
        ipsum: { type: 'number', gte: 0, lte: 5 }, // Between [0; 5]
        dolor: { type: 'number', eq: [0, 3, 6, 9] }, // Equal to 0, 3, 6 or 9
        sit: { type: 'number', ne: [0, 3, 6, 9] }, // Not equal to 0, 3, 6 nor 9
    },
```

```
};

var c1 = { lorem: 3, ipsum: 0, dolor: 6, sit: 2 };
var c2 = { lorem: 0, ipsum: -1, dolor: 5, sit: 3 };

inspector.validate(schema, c1); // Valid
inspector.validate(schema, c2); // Invalid
```

someKeys

- **type**: array of string.
- **usable on**: object.

Check whether one of the given keys exists in object (useful when they are optional).

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    someKeys: ['lorem', 'ipsum'],
    properties: {
        lorem: { type: 'any', optional: true },
        ipsum: { type: 'any', optional: true },
        dolor: { type: 'any' },
    },
};

var c1 = { lorem: 0, ipsum: 1, dolor: 2 };
var c2 = { lorem: 0, dolor: 2 };
var c3 = { dolor: 2 };

inspector.validate(schema, c1); // Valid
inspector.validate(schema, c2); // Valid
inspector.validate(schema, c3); // Invalid: Neither @.lorem nor @.ipsum is in c3.
```

strict

- **type**: boolean.
- **default**: false.
- **usable on**: object.

Only keys provided in field "properties" may exist in the object. Strict will be ignored if properties has the special key '*'.

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    strict: true,
    properties: {
        lorem: { type: 'any' },
        ipsum: { type: 'any' },
        dolor: { type: 'any' },
    },
};

var c1 = { lorem: 0, ipsum: 1, dolor: 2 };
var c2 = { lorem: 0, ipsum: 1, dolor: 2, sit: 3 };

inspector.validate(schema, c1); // Valid
inspector.validate(schema, c2); // Invalid: @.sit should not exist.
```

exec

- **type**: function, array of function.
- **usable on**: any.

Custom checker =). "exec" functions take two three parameter (schema, post [, callback]). To report an error, use `this.report([message], [code])`. Very useful to make some custom validation.

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    properties: {
        lorem: {
            type: 'number',
            exec: function (schema, post) {
                // here schema === schema.properties.lorem and post === @.lorem
                if (post === 3) {
                    // As soon as `this.report()` is called, candidate is not valid.
                    this.report('must not equal 3 =('); // Ok...it's exactly like "ne: 3"
                }
            },
        },
    },
};

var c1 = { lorem: 2 };
var c2 = { lorem: 3 };

inspector.validate(schema, c1); // Valid
inspector.validate(schema, c2); // Invalid: "@.lorem must not equal 3 =(".
```

properties

- **type**: object.
- **usable on**: object.

For each property in the field "properties", whose value must be a schema, validation is called deeper in object.

The special property '*' is validated against any properties not specifically listed.

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    properties: {
        lorem: {
            type: 'object',
            properties: {
                ipsum: {
                    type: 'object',
                    properties: {
                        dolor: { type: 'string' },
                    },
                },
            },
        },
        consectetur: { type: 'string' },
        '*': { type: 'integer' },
    },
};

var c1 = {
    lorem: {
        ipsum: {
            dolor: 'sit amet',
        },
    },
    consectetur: 'adipiscing elit',
    adipiscing: 12,
};
```

```
var c2 = {
    lorem: {
        ipsum: {
            dolor: 12,
        },
    },
    consectetur: 'adipiscing elit',
};

inspector.validate(schema, c1); // Valid
inspector.validate(schema, c2); // Invalid: @.lorem.ipsum.dolor must be a string.
```

items

- **type**: object, array of object.
- **usable on**: array.

Allow to apply schema validation for each element in an array. If it's an object, then it's a schema which will be used for all the element. If it's an array of object, then it's an array of schema and each element in an array will be checked with the schema which has the same position in the array.

Example

```
var inspector = require('schema-inspector');

var schema1 = {
    type: 'array',
    items: { type: 'number' },
};

var schema2 = {
    type: 'array',
    items: [{ type: 'number' }, { type: 'number' }, { type: 'string' }],
};

var c1 = [1, 2, 3];
var c2 = [1, 2, 'string!'];

inspector.validate(schema1, c1); // Valid
inspector.validate(schema1, c2); // Invalid: @[2] must be a number.
inspector.validate(schema2, c1); // Invalid: @[2] must be a string.
inspector.validate(schema2, c2); // Valid
```

alias

- **type**: string.
- **usable on**: any.

Allow to display a more explicit property name if an error is encounted.

Example

```
var inspector = require('schema-inspector');

var schema1 = {
    type: 'object',
    properties: {
        _id: { type: 'string' },
    },
};

var schema2 = {
    type: 'object',
    properties: {
        _id: { alias: 'id', type: 'string' },
    },
};

var c1 = { _id: 1234567890 };

var r1 = inspector.validate(schema1, c1);
```

```
var r2 = inspector.validate(schema2, c1);
console.log(r1.format()); // Property @._id: must be string, but is number
console.log(r2.format()); // Property id (@._id): must be string, but is number
```

error

- **type**: string.
- **usable on**: any.

This field contains a user sentence for displaying a more explicit message if an error is encounted.

Example

```
var inspector = require('schema-inspector');

var schema1 = {
    type: 'object',
    properties: {
        _id: { type: 'string' },
    },
};

var schema2 = {
    type: 'object',
    properties: {
        _id: { type: 'string', error: 'must be a valid ID.' },
    },
};

var c1 = { _id: 1234567890 };

var r1 = inspector.validate(schema1, c1);
var r2 = inspector.validate(schema2, c1);
console.log(r1.format()); // Property @._id: must be string, but is number.
console.log(r2.format()); // Property @._id: must be a valid ID.
```

code

- **type**: string.
- **usable on**: any.

This field contains a user code for displaying a more uniform system to personnalize error message.

Example

```
var inspector = require('schema-inspector');

var schema1 = {
    type: 'object',
    properties: {
        _id: { type: 'string' },
    },
};

var schema2 = {
    type: 'object',
    properties: {
        _id: { type: 'string', code: 'id-format' },
    },
};

var c1 = { _id: 1234567890 };

var r1 = inspector.validate(schema1, c1);
var r2 = inspector.validate(schema2, c1);
console.log(r1.error[0].code); // null
console.log(r2.error[0].code); // 'id-format'
```

Sanitization

type

- **type**: string.
- **usable on**: any.
- **possible values**
  - `number`
  - `integer`
  - `string`
  - `boolean`
  - `date` (constructor === Date)
  - `object` (constructor === Object)
  - `array` (constructor === Array)

Cast property to the given type according to the following description:

- **to number from**:
  - string (ex: "12.34" -> 12.34) 

  - date (ex: new Date("2014-01-01") -> 1388534400000)


- **to integer from**:
  - number
    - 12.34 -> 12
  - string
    - "12.34" -> 12
  - boolean
    - true -> 1
    - false -> 0
  - date
    - new Date("2014-01-01") -> 1388534400000
- **to string from**:
  - boolean
    - true -> "true"
  - number
    - 12.34 -> "12.34"
  - integer
    - 12 -> "12"
  - date
    - new Date("2014-01-01") -> "Wed Jan 01 2014 01:00:00 GMT+0100 (CET)"
  - array
    - [12, 23, 44] -> '12,34,45'
    - To join with a custom string, use **joinWith** key (example: { type: "string", joinWith: "|" } will transform [12, 23, 44] to "12|23|44").
- **to date from**:
  - number / integer
    - 1361790386000 -> Wed Jan 01 2014 01:00:00 GMT+0100 (CET)
  - string
    - "2014-01-01 -> Wed Jan 01 2014 01:00:00 GMT+0100 (CET)
    - "Wed Jan 01 2014 01:00:00 GMT+0100 (CET)" -> Wed Jan 01 2014 01:00:00 GMT+0100 (CET)
- **to object from**:
  - string
    - '{"love":"open source"}' -> { love: "open source" }
- **to array from**:
  - string ("one,two,three" -> ["one", "two", "three"], '[1,"two",{"three":true}]' -> [1, 'two', { three: true }])
  - anything except undefined and array (23 -> [23])
  - To split with a custom string (other than ","), use the key **splitWith** (example: { type: "array", splitWith: "|"" } will transform "one|two|three" to ["one", "two", "three"]).*

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'array',
    items: { type: 'string' },
};

var c = [12.23, -34, true, false, 'true', 'false', [123, 234, 345], { obj: "yes" }];
```

```
var r = inspector.sanitize(schema, c);
/*
r.data: [ '12.23', '-34', 'true', 'false', 'true', 'false', '123,234,345', '{"obj":"yes"}' ]
*/
```

def

- **type**: any.
- **usable on**: any.

Define default value if property does not exist, or if type casting is to fail because entry type is not valid (cf type).

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    properties: {
        lorem: { type: 'number', def: 10 },
        ipsum: { type: 'string', def: 'NikitaJS', optional: false },
        dolor: { type: 'string' },
    },
};

var c = {
    lorem: [12, 23], // convertion to number is about to fail
    // (array -> number is not possible)
    // ipsum is not privided
    dolor: 'sit amet', // "dolor" is already a string
};

var r = inspector.sanitize(schema, c);
/*
r.data: {
    lorem: 10,
    ipsum: 'NikitaJS',
    dolor: 'sit amet'
}
*/
```

optional

- **type**: boolean.
- **default**: true.
- **usable on**: any.

Property is set to `schema.def` if not provided and if optional is `false` .

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    properties: {
        lorem: { type: 'number', optional: false, def: 12 },
        ipsum: { type: 'string', optional: true, def: 23 },
        dolor: { type: 'string', def: 'NikitaJS', def: 34 }, // (optional: true)
    },
};

var c = {};

var r = inspector.sanitize(schema, c);
/*
r.data: {
    lorem: 12 // Only lorem is set to 12 because it is not optional.
```

```
}
*/
```

rules

- **type**: string, array of string.
- **usable on**: string.
- **possible values**:
  - `upper` : Every character will be changed to uppercase.
  - `lower` : Every character will be changed to lowercase.
  - `title` : For each word (/\S*/g), first letter will be changed to uppercase, and the rest to lowercase.
  - `capitalize` : Only the first letter of the string will be changed to uppercase, the rest to lowercase.
  - `ucfirst` : Only the first letter of the string will be changed to uppercase, the rest is not modified.
  - `trim` : Remove extra spaces.

Apply the given rule to a string. If several rules are given (array), then they are applied in the same order than in the array.

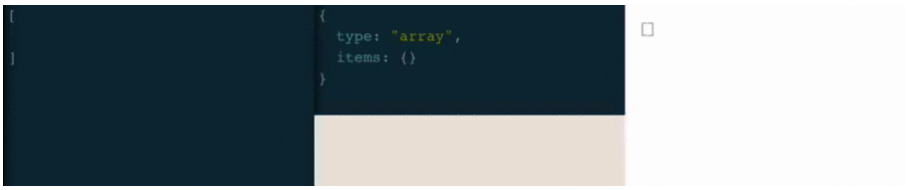
Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    properties: {
        lorem: { type: 'string', rules: 'upper' },
        ipsum: { type: 'string', rules: ['trim', 'title'] },
    },
};

var c = {
    lorem: ' tHiS is sParTa! ',
    ipsum: '   tHiS is sParTa!    ',
};

var r = inspector.sanitize(schema, c);
/*
r.data: {
    lorem: ' THIS IS SPARTA! ',
    ipsum: 'This Is Sparta!' // has been trimed, then titled
}
*/
```

min, max

- **type**: string, number.
- **usable on**: string, number.

Define minimum and maximum value for a property. If it's less than minimum, then it's set to minimum. If it's greater than maximum, then it's set to maximum.



Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'array',
```

```
    items: { type: 'number', min: 10, max: 20 },
};

var c = [5, 10, 15, 20, 25];

var r = inspector.sanitize(schema, c);
/*
 r.data: [10, 10, 15, 20, 20]
 c[0] (5) was less than min (10), so it's been set to 10.
 c[4] (25) was greater than max (20), so it's been set to 20.
*/
```

---

minLength, maxLength

- **type**: integer.
- **usable on**: string.

Adjust string length to the given number.

TODO: We must be able to choose which character we want to fill the string with.

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'array',
    items: { type: 'string', minLength: 8, maxLength: 11 },
};

var c = ['short', 'mediumSize', 'tooLongForThisSchema'];

var r = inspector.sanitize(schema, c);
/*
 r.data: ['short---', 'mediumSize', 'tooLongForT']
*/
```

---

strict

- **type**: boolean.
- **default**: false.
- **usable on**: any.

Only key provided in field "properties" will exist in object, others will be deleted.

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    strict: true,
    properties: {
        good: { type: 'string' },
    },
};

var c = {
    good: 'yes',
    bad: 'nope',
};

var r = inspector.sanitize(schema, c);
/*
r.data: {
    good: 'yes'
}
*/
```

---

exec

- **type**: function, array of functions.
- **usable on**: any.

Custom checker =). "exec" functions take two three parameter (schema, post [, callback]), and must return the new value. To report an sanitization, use `this.report([message])`. Very useful to make some custom sanitization.

NB: If you don't want to return a differant value, simply return `post`, do not return nothing (if you do so, the new value will be `undefined`).

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'array',
    items: {
        type: 'string',
        exec: function (schema, post) {
            if (typeof post === 'string' && !/^nikita$/i.test(post)) {
                this.report();
                return '_INVALID_';
            }
            return post;
        },
    },
};

var c = ['Nikita', 'lol', 'NIKITA', 'thisIsGonnaBeSanitized!'];

var r = inspector.sanitize(schema, c);
/*
 r.data: [ 'Nikita', '_INVALID_', 'NIKITA', '_INVALID_' ]
*/
```

properties

- **type**: object.
- **usable on**: object.

Work the same way as **validation "properties"**.

items

- **type**: object, array of object.
- **usable on**: array.

Work the same way as **validation "items"**.

Custom fields

punctual use

When you need to use the same function in `exec` field several time, instead of saving the function and declaring `exec` several times, just use custom field. First you have to provide a hash containing a function for each custom field you want to inject. Then you can call them in your schema with $"your field name". For example if you provide a custom field called "superiorMod", you can access it with name "$superiorMod".

Example

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    properties: {
        lorem: { type: 'number', $divisibleBy: 5 },
        ipsum: { type: 'number', $divisibleBy: 3 },
    },
};

var custom = {
    divisibleBy: function (schema, candidate) {
        var dvb = schema.$divisibleBy;
        if (candidate % dvb !== 0) {
```

```
            this.report('must be divisible by ' + dvb);
        }
    },
};

var c = {
    lorem: 10,
    ipsum: 8,
};
inspector.validate(schema, candidate, custom); // Invalid: "@.ipsum must be divisible by 3"
```

extension

Sometime you want to use a custom field everywhere in your program, so you may extend Schema-Inspector to do so. Just call the method *inspector.Validation.extend(customFieldObject)* or *inspector.Sanitization.extend(customFieldObject)*. If you want to reset, simply call *inspector.Validation.reset()* or *inspector.Sanitization.reset()*. You also can remove a specific field by calling *inspector.Validation.remove(field)* or *inspector.Sanitization.remove(field)*.

Example

```
var inspector = require('schema-inspector');

var custom = {
    divisibleBy: function (schema, candidate) {
        var dvb = schema.$divisibleBy;
        if (candidate % dvb !== 0) {
            this.report('must be divisible by ' + dvb);
        }
    },
};

var schema = {
    type: 'object',
    properties: {
        lorem: { type: 'number', $divisibleBy: 5 },
        ipsum: { type: 'number', $divisibleBy: 3 },
    },
};

inspector.Validation.extend(custom);

var candidate = {
    lorem: 10,
    ipsum: 8,
};

inspector.validate(schema, candidate);
/*
 As you can see, no more object than schema and candidate has been provided.
 Therefore we can use `$divisibleBy` everywhere in all schemas, for each
 inspector.validate() call.
*/
```

Context

Every function you declare as a custom parameter, or with `exec` field will be called with a context. This context allows you to access properties, like `this.report()` function, but also `this.origin`, which is equal to the object sent to `inspector.validate()` or `inspector.sanitize()`.

Example

```
// ...
var schema = { /* ... */ };
var custom = {
    divisibleBy: function (schema, candidate) {
        // this.origin === [12, 23, 34, 45]
        // ...
    },
};
var candidate = [12, 23, 34, 45];
```

```
var result = inspector.validate(schema, candidate, custom);
// ...
```

Asynchronous call

How to

All of the examples above used synchronous calls (the simplest). But sometimes you want to call validation or sanitization asynchronously, in particular with `exec` and custom fields. It's pretty simple: To do so, just send a callback as extra parameter. It takes 2 parameters: error and result. Actually Schema-Inspector should send back no error as it should not throw any if called synchronously. But if you want to send back and error in your custom function, inspection will be interrupted, and you will be able to retrieve it in your callback.

You also have to declare a callback in your `exec` or custom function to make Schema-Inspector call it asynchronously, else it will be call synchronously. That means you may use `exec` synchronous function normally even during and asynchronous call.

Example

```
var inspector = require('schema-inspector');

var schema = { /* ... */ };
var candidate = { /* ... */ };

inspector.validate(schema, candidate, function (err, result) {
    console.log(result.format());
});
```

Example with custom field

```
var inspector = require('schema-inspector');

var schema = { /* ... */ };
var candidate = { /* ... */ };
var custom = { /* ... */ };

inspector.validate(schema, candidate, custom, function (err, result) {
    console.log(result.format());
});
```

Here is a full example where you may have to use it:

```
var inspector = require('schema-inspector');

var schema = {
    type: 'object',
    properties: {
        lorem: { type: 'number', $divisibleBy: 4 },
        ipsum: { type: 'number', $divisibleBy: 5 },
        dolor: { type: 'number', $divisibleBy: 0, optional: true },
    },
};

var custom = {
    divisibleBy: function (schema, candidate, callback) {
        // Third parameter is declared:
        // Schema-Inspector will wait this function to call this `callback` to keep running.
        var dvb = schema.$divisibleBy;
        if (typeof dvb !== 'number' || typeof candidate !== 'number') {
            return callback();
        }
        var self = this;
        process.nextTick(function () {
            if (dvb === 0) {
                return callback(
                    new Error('Schema error: Divisor must not equal 0')
                );
            }
            var r = candidate / dvb;
            if ((r | 0) !== r) {
                self.report('should be divisible by ' + dvb);
            }
```

```
            callback();
        });
    },
};

var candidate = {
    lorem: 12,
    ipsum: 25,
};

inspector.validate(schema, candidate, custom, function (err, result) {
    console.log(result.format());
});
```

Keywords

validation sanitization inspector validator json validate sanitize

Install

```
> npm i schema-inspector
```

Repository

github.com/schema-inspector/schema-inspector

Homepage

schema-inspector.github.io/schema-inspector/

Weekly Downloads

7,091

Version	License
2.0.2	MIT
Unpacked Size	**Total Files**
96.2 kB	6
Issues	**Pull Requests**
5	5

Last publish

a month ago

Collaborators


>_Try on RunKit

Report malware

Support

Help

Advisories

Status

Contact npm

Company

About

Blog

Press

Terms & Policies

Policies

Terms of Use

Code of Conduct

Privacy