Prudence

zkldi

Feb 20, 2021

VALIDATOR:

Prudence is a simple JS object validator.

The source code can be found here.

Prudence is available for download on npm.

The current version for Prudence is `v0.3.0`.

CHAPTER

ONE

DEFAULT VALIDATOR

The Prudence validator can be accessed by simply calling `Prudence()` with its arguments.

Prudence (*object*, *schema*[, *errorMessages*[, *options*]])

Validated the provided object against the provided schema.

Arguments

- **object** (*object*) – The object to validate.
- **schema** (*object*) – The schema to validate against.
- **errorMessages** (*object*) – Error messages to use on error, defaults to `{}`.
- **options** (*object*) – Options for the Prudence validator. See *Options*.

Returns A `string` error message on failure, and `null` on success.

Warning: Prudence returns `null` on success. To keep this clear, you **should** use Prudence like this:

```
let err = Prudence(/* ... */);

if (err) { /* do error handling */ }
```

1.1 Instanced Validator

In the case that you want to always use a specific set of options while parsing, you can instantiate your own validator with the following command.

```
let myPrudenceValidator = new Prudence.Validator(myPreferedOptions);

// Note: you can still pass an options param, all this accomplishes is changing the␣
↪validator defaults.
let err = myPrudenceValidator.validate(object, schema, errorMessages, options);
```

1.2 Error Message Overrides

The third argument to Prudence is `errorMessages`, and it expects an object with identical structure to the schema, containing error messages.

This can be used to override existing error messages, or to provide them for functions (Although the alternative of Validation Functions is prefered).

```
let schema = {
    username: "string"
}

let errMsgs = {
    username: "Custom Error Message Here"
}

// err is "[username] Custom Error Message Here. Received null." - Note the automatic␣
↪prefixing and suffixing of information.
let err = Prudence({ username: null }, schema, errMsgs);
```

1.3 Options

The below table indicates the options available for Prudence. The defaults are in brackets.

Key	Type (Default)	Description
`allowExcessKeys`	`boolean (false)`	Whether or whether not to allow excess keys on the provided object, such as { foo, bar } being validated by schema { foo }.
`throwOnNonObject`	`boolean (true)`	Whether or whether not to throw an error if a non-object is provided as the object parameter (such as accidentally sending `undefined`).

1.4 Default Options

The default options for Prudence can be accessed at `Prudence.Validator.defaultOptions`.

CHAPTER

TWO

PRUDENCE SCHEMA

2.1 Declaration

A schema is just an object. You can create one as you would any other object.

```
let schema = {
    username: "string"
};
```

The above schema declaration asserts that we expect an object with the key `username`, and for that username to also be a string.

2.2 String Literal

The first way of defining an expectation is to write a string. Strings are used by checking the `typeof` the given value equals the string.

The valid values for this can be found at MDN.

```
let schema = {
    username: "string",
    xp: "number",
    bigInt: "bigint"
}

// is valid
let err = Prudence({ username: "zkldi", xp: 1234.567, bigInt: 123n }, schema);

// is invalid: err2 is "[username] Expected typeof string. Received null."
let err4 = Prudence({ username: null, xp: 1234.567, bigInt: 123n }, schema);
```

Warning: **Huge Exception**. For usefulness's sake, Prudence overasserts that typeof null === "null", and **NOT** "object".

2.2.1 Nullable Fields

You can prefix your strings with `?`, which makes the field nullable. As an example,

```
let schema = {
    username: "?string"
}

// is valid.
let err = Prudence({ username: "zkldi" }, schema);

// is also valid.
let err2 = Prudence({ username: null }, schema);
```

2.2.2 Optional Fields

You can also prefix your strings with `*`, which makes the field optional. As an example,

```
let schema = {
    username: "*string"
}

// is valid.
let err = Prudence({ username: "zkldi" }, schema);

// is also valid.
let err2 = Prudence({ }, schema);

// is also valid (the two are essentially equivalent).
let err3 = Prudence({ username: undefined }, schema);
```

You can use both of these by prefixing with `*?`.

Note: The order must be `*?`, `?*` will throw an error.

2.2.3 Error Messages

Prudence is able to automatically create meaningful error messages from this schema type.

The error message for this is as follows:

```
[(location of error)]:  Expected typeof (schemaValue) [or null, or no value].
Received (objectValue).
```

2.3 Validation Functions

The other option inside a schema is to use a function. If a value in a schema is a function, it is called with up to two arguments.

It is expected that this function returns a boolean value, whether the input is valid or not.

The first argument is the object's value at this same key.

```
let schema = {
    age: (self) => Number.isSafeInteger(self) && self >= 18
}

// is valid!
let err = Prudence({ age: 19 }, schema);
```

The second argument is the object the value was found inside. This is often not that useful, but can be used for conditional validity, such as when one keys validity depends on another.

In the below example, we define a schema that checks if an objects `end` key is greater than its `start` key.

```
let schema = {
    start: "number",
    end: (self, parent) => typeof self === "number" && self > parent.start
}

// is valid.
let err = Prudence({ start: 14, end: 190 }, schema);

// is invalid, but we'll get to what err2 is in a second.
let err2 = Prudence({ start: 14, end: 13 }, schema);
```

2.3.1 Function Error Messages

Prudence is **NOT** able to construct meaningful error messages from functions.

To enable Prudence to construct a meaningful error message, you attach the property `errorMessage` to the function.

As an example:

```
let fn = (self) => self === "hello world!";

fn.errorMessage = "Expected the string \"hello world!\"";
```

Given that this is rather cumbersome (and difficult in languages like TypeScript), Prudence also provides a utility to create these functions.

```
let fn = Prudence.createFn((self) => self === "hello world!", "Expected the string \
↪"hello world!\"");
```

The two examples have identical results.

Note: These are referred to as Validation Functions in this documentation, and more information can be found here, at *Validation Function*.

The other way to pass an error message is to pass a third argument to Prudence - See *Error Message Overrides*.

2.4 Nesting

Prudence is natively nestable, which means if you assign an object to a schema's value, it will simply work recursively.

```
let schema = {
    a: {
        b: {
            c: {
                "user-name": "string"
            }
        }
    }
}

// is valid!
let err = Prudence({a:{b:{c:{"user-name": "zkldi"}}}}), schema);

// Prudence also automatically provides meaningful information about where an error␣
↪occured
// even if nested!

// "[a.b.c["user-name"]]: Expected typeof string. Received null."
let err2 = Prudence({a:{b:{c:{"user-name": null}}}}, schema);
```

Prudence will throw an error if an object does not match the "shape" of the schema, as follows:

```
let schema = {
    a: {
        b: {
            c: {
                "user-name": "string"
            }
        }
    }
}

// "[a.b.c]: Object does not match structure of schema, expected this location to␣
↪have an object."
let err = Prudence({
    a: {
        b: {
            c: undefined
        }
    }
}), schema);
```

Note: Prudence traverses the schema, not the object. You do not need to worry about deeply nested objects being passed.

2.5 Arrays

If a value in a schema is an array with a single value, Prudence will assume you want to validate an array against that single value.

This is perhaps clearer with examples:

```
let schema = {
    username: "string",
    aliases: ["string"]
}

// is valid
let err = Prudence({ username: "zkldi", aliases: ["foo", "bar"] }, schema);

let functionSchema = {
    username: "string",
    friendIDs: [(self) => Number.isSafeInteger(self)]
}

// is valid
let err2 = Prudence({ username: "zkldi", friendIDs: [1,2,3,4] }, schema);

let complexSchema = {
    username: "string",
    groupchats: [
        {
            name: "string",
            members: [Prudence.isInteger],
        },
    ]
}

// is valid
let err3 = Prudence({
    username: "zkldi",
    groupchats: [
        {
            name: "the boys",
            members: [13, 14, 15],
        },
        {
            name: "the fellas",
            members: [13, 16, 17],
        }
    ]
}, schema);
```

Prudence can also infer where an error occured inside an array, as follows:

```
let schema = {
    username: "string",
    aliases: ["string"]
}

// [aliases[1]] Expected typeof string. Received null.
let err = Prudence({ username: "zkldi", aliases: ["foo", null, "bar" ] }, schema);
```

2.6 Built-ins

Prudence comes with "built-in" functions that satisfy common use cases for validation. You can see documentation for these at *Built-In Functions*.

CHAPTER

THREE

VALIDATION FUNCTION

The function expected by Prudence for validation.

(*[self[, parent]]*)

Arguments

- **self** (*any*) – The object's value for this key. This can be anything.
- **parent** (*object*) – The object the above value came from.

Attribute string errorMessage An error message for Prudence to display if this function returns false.

Returns boolean

Note: This function is allowed to return truthy/falsy values, but it is not prefered.

CHAPTER

FOUR

BUILT-IN FUNCTIONS

All built in functions have built in error messages.

If you think this list should be expanded, feel free to submit a feature request!

4.1 Validation Functions

The below functions should be used like:

```
let schema = {
    key: Prudence.isInteger
}
```

`Prudence.`**`isInteger`**
Checks whether a value is a safe integer or not.

> **Note:** This is an alias for Number.isSafeInteger.

`Prudence.`**`isPositiveInteger`**
Checks whether a value is a positive integer or not. 0 is regarded as positive.

`Prudence.`**`isPositiveNonZeroInteger`**
Checks whether a value is a positive, non-zero integer or not.

4.2 Validation Function Generators

The below functions should be used as follows:

```
let schema = {
    key: Prudence.isBoundedInteger(1, 10)
}
```

This is because these are functions that return validation functions, rather than simply being validation functions.

The arguments they expect are detailed in their below definitions.

`Prudence.`**`isIn`**(*...values*)
Returns a function that checks whether the value given is inside the list of arguments.

Arguments

- **`...values`** – Rest Parameter: the values to check against.

Returns *Validation Function*

Example usage:

```
let schema = {
    fruit: Prudence.isIn("apple", "banana"),
    alternatively: Prudence.isIn(["apple", "banana"])
}
```

Note: You can also pass an array as a first and only argument, and it will automatically be expanded.

`Prudence.`**`isBoundedInteger`**(*lower*, *upper*)

Returns a function that checks whether a value is an integer and between the two arguments.

This is inclusive on both ends.

Arguments

- **`lower`** (*`integer`*) – The number the object's value must be greater than or equal to.
- **`upper`** (*`integer`*) – The number the object's value must be less than or equal to.

Returns *Validation Function*

`Prudence.`**`isBoundedString`**(*lower*, *upper*)

Returns a function that checks whether a value is a string and its length is between the two arguments.

This is inclusive on both ends.

Arguments

- **`lower`** (*`integer`*) – The number the string's length must be greater than or equal to.
- **`upper`** (*`integer`*) – The number the string's length must be less than or equal to.

Returns *Validation Function*

`Prudence.`**`regex`**(*regex*)

Returns a function that checks whether the input is a string and matches the given regex.

This exists because regex.test coerces input to a string, even when thats not what we want.

Arguments

- **`regex`** (*`regex`*) – The number the string's length must be greater than or equal to.

Returns *Validation Function*

Warning: The below functions exist because the <, <=, > and >= operators in JS are not strict (think ==) and convert non-numeric input into numbers, when that's almost certainly not what you'd want.

`Prudence.`**`gt`**(*num*)

Returns a function that checks whether a value is a number and greater than the argument.

Arguments

- **`num`** (*`number`*) – The number the object's value must be greater than.

Returns *Validation Function*

`Prudence.`**`gte`**(*num*)

Returns a function that checks whether a value is a number and greater than or equal to the argument.

Arguments

- **num** (*number*) – The number the object's value must be less than or equal to.

Returns *Validation Function*

Prudence.**lt**(*num*)
Returns a function that checks whether a value is a number and less than the argument.

Arguments

- **num** (*number*) – The number the object's value must be greater than.

Returns *Validation Function*

Prudence.**lte**(*num*)
Returns a function that checks whether a value is a number and less than or equal to the argument.

Arguments

- **num** (*number*) – The number the object's value must be less than or equal to.

Returns *Validation Function*

Prudence.**gtInt**(*num*)
Returns a function that checks whether a value is an integer and greater than the argument.

Arguments

- **num** (*number*) – The number the object's value must be greater than.

Returns *Validation Function*

Prudence.**gteInt**(*num*)
Returns a function that checks whether a value is an integer and greater than or equal to the argument.

Arguments

- **num** (*number*) – The number the object's value must be less than or equal to.

Returns *Validation Function*

Prudence.**ltInt**(*num*)
Returns a function that checks whether a value is an integer and less than the argument.

Arguments

- **num** (*number*) – The number the object's value must be greater than.

Returns *Validation Function*

Prudence.**lteInt**(*num*)
Returns a function that checks whether a value is an integer and less than or equal to the argument.

Arguments

- **num** (*number*) – The number the object's value must be less than or equal to.

Returns *Validation Function*

CHAPTER

FIVE

CREATE VALIDATION FUNCTION

Prudence.**CreateFn**(*fn*, *errMsg*)

Creates a *Validation Function*.

Arguments

- **fn** (*function*) – The function to use.
- **errMsg** (*string*) – An error message to attach.

Returns *Validation Function*.

CHAPTER

SIX

EXPRESS MIDDLEWARE

This provides utilities for using Prudence in Express.

`Prudence.`**`Middleware`**(*schema*[, *errorHandler*[, *errorMessages*[, *options*]]])

Creates an Express Middleware that validates req.query (method "GET") or req.body (all other methods) using Prudence.

On success, this

Arguments

- **`schema`** (*`object`*) – The Prudence schema to validate req.query or req.body against.
- **`errorHandler`** (*`function`*) – If prudence hits an error, this function determines what to do with it. See *Error Handler*, optional.
- **`errorMessages`** (*`object`*) – Error message overrides for Prudence, optional.
- **`options`** (*`object`*) – The options for Prudence, optional.

Returns Express Middleware Function

Note: Passing an errorHandler is optional. If one is not passed, the middleware will return 400 and the error message in JSON.

Example usage:

```
let schema = {
    username: Prudence.isBoundedString(3, 20),
    password: Prudence.isBoundedString(3, 20),
    confirmPassword: (self, parent) => self === parent.password
};

let errorHandler = (req, res, next, errMsg) => res.send(400).send(`You messed up! $
↪{errMsg}`);

router.post("/register", Prudence.Middleware(schema, errorHandler), (req, res) => {
    // assign cookies or something
    return res.redirect("/"); // send users back to homepage
});
```

6.1 Error Handler

A function which is passed 4 parameters. In order; `req`, `res`, `next`, `errMsg`.

The first three are identical to their Express middleware counterparts. The errMsg is the string Prudence returned.

CHAPTER

SEVEN

CURRY MIDDLEWARE

It's common that you'll want to re-use error handlers. To support this, you can use the following function.

`Prudence.`**`CurryMiddleware`**`(`*errorHandler*`)`

Returns a function that takes schema, [errorMessages, [options]] to create express middleware. The initially passed errorHandler is always used.

Arguments

- **`errorHandler`** (*`function`*) – If prudence hits an error, this function determines what to do with it. See *Error Handler*, optional.

Returns See above.

Example usage:

```
let schema = {
    username: Prudence.isBoundedString(3, 20),
    password: Prudence.isBoundedString(3, 20),
    confirmPassword: (self, parent) => self === parent.password
};

let errorHandler = (req, res, next, errMsg) => res.send(400).send(`You messed up! $
↪{errMsg}`);

let MiddlewareGenerator = Prudence.CurryMiddleware(errorHandler);

router.post("/register", MiddlewareGenerator(schema), (req, res) => {
    // assign cookies or something
    return res.redirect("/"); // send users back to homepage
});
```

Example usage:

```
let schema = {
    username: Prudence.isBoundedString(3, 20),
    password: Prudence.isBoundedString(3, 20),
    confirmPassword: (self, parent) => self === parent.password
};

let errorHandler = (req, res, next, errMsg) => res.send(400).send(`You messed up! $
↪{errMsg}`);

router.post("/register", Prudence.Middleware(schema, errorHandler), (req, res) => {
    // assign cookies or something
    return res.redirect("/"); // send users back to homepage
});
```

INDEX

Symbols

C

G

I

L

M

P

R