Introduction

`super-easy-validator` a super simple npm package with zero dependency which helps in data validation in much simpler way. Its inspired by laravel validator, but even better. Its even more powerful than express-validator and zod.

Please write any issues on github if you found any. Don't hesitate to suggest any new features.

Useful Links

How to install

Run this command in the root of your npm project

```
npm i super-easy-validator
```

Usage

1. Basic Usage Step-By-Step

Suppose we need to validate the following data:

```
const data = {
  name: 'test123',
  gender: 'Male',
  adult: true,
  id: '123e4567-e89b-12d3-a456-426655440000',
  creditCard: '1987654312345678',
  isMarried: 'no',
  profile: 'example.com',
  password: 'ab',
  favoriteFoods: ['chicken', 'egg roll', 'french fries'],
  rating: 4.5,
  ratings: [3, 5, undefined, true, 5.67],
  score: 234.5,
  accountBalance: 100.345,
  hash: 'a6g8d7Fkf9Du',
  hash2: 'PDH78DI908g56',
  serverIp: '8.45.23.0',
  dob: '1996-01-10T23:50:00.0000+05:30',
  time: '23:50',
  address: {
    pin: '829119',
    city: 'Rock Port',
    country: {
      code: 'IN',
    }
  },
  users: [
    {
      name: 'John Doe',
      age: 20,
      gender: 'male',
    },
    { }
  ],
  limit: '90',
}
```

Here is the step by step guide to validate the above data using this package:

Step 1: Creating validations rules

At first we need to create the validation rules which is just a `Record` of key value pairs. This validation rules object will later be used for validating data.

```
let rules = {
    mail: 'optional|email',
    phone: 'optional|phone',
    $atleast: 'mail|phone',
    $atmost: 'mail|phone|size:1',
    name: 'name|field:person name',
    gender: 'enums:male,female',
    adult: 'enums:true,false',
    id: 'uuid',
    creditCard: 'string|regex:/^[0-9]{16}$/',
    isMarried: 'boolean',
    userId: 'mongoid',
    profile: 'url',
    password: 'string|min:3|max:15',
    favoriteFoods: 'array|min:3|max:6',
    rating: 'number|enums:1,2,3,4,5|error:rating is not correct, please fix it',
    ratings: 'arrayof:optional|arrayof:natural|arrayof:max:5|field:ratingsList',
    score: 'number|whole',
    accountBalance: 'number|min:0|decimalsize:2',
    hash: 'lower',
    hash2: 'upper',
    serverIp: 'ip',
    dob: 'date',
    time: 'time',
    address: {
        pin: 'string|natural|size:6',
        city: 'name',
        country: {
            code: 'alpha|upper|size:2',
        },
    },
    users: [
        {
            name: 'name',
            age: 'natural',
            gender: 'enums:male,female',
        },
    ],
    person: 'object',
    'person.address': 'string',
    limit: 'optional|string|natural|min:100',
}
```

Step 2: Validating data

Then, we can validate our data using the rules created in previous step:

```
```

```
const Validator = require('super-easy-validator');

let { errors } = Validator.validate(rules, data);
if (errors) {
  console.log(errors);
}
```

Final output

This is how the final output should look like:

```
[
  'at least one of `mail` and `phone` is required',
  ``person name` must be a valid name',
  ``gender` is invalid',
  ``isMarried` must be a valid boolean',
  ``userId` is required',
  ``profile` must be a valid url',
  ``password` must have length of at least 3',
  'rating is not correct, please fix it',
  ``ratingsList[3]` must be a valid number',
  ``ratingsList[4]` must be a valid natural number',
  ``score` must be a valid whole number',
  ``accountBalance` must have 2 decimal places',
  ``hash` must not contains upper case letters',
  ``hash2` must not contains lower case letters',
  ``users[1].name` is required',
  ``users[1].age` is required',
  ``users[1].gender` is required',
  ``person` is required',
  ``person.address` is required',
  ``limit` must be at least 100',
]
```

> **NOTE:** You could also have written:

```
let { errors } = Validator.validate(rules, data, {quotes: 'backtick'});
```

> If you want quotes to be added around each field names in error messages. For more info, see quotes API section.

2. Data Validation of request query in express

Suppose we need to validate some fields which are all present in `req.query` (request query) in express GET API. As you already know every fields in `req.query` is string. Also in most cases these fields are optional, so we have to make sure to consider these things before writing validation. Here is a proper example.

```
const { validate } = require('super-easy-validator');

function validateGetProducts(req, res, next) {
  try {
    let rules = {
      limit: 'optional|string|natural|max:100',
      page: 'optional|string|natural',
      searchKey: 'optional|string|min:1',
      productId: 'optional|mongoid',
      withProduct: 'optional|string|boolean',
      expiryMin: 'optional|date',
      expiryMax: 'optional|date',
      priceMin: 'optional|string|positive',
      priceMax: 'optional|string|positive',
      createdAtMin: 'optional|date',
      createdAtMax: 'optional|date',
      sortBy: 'optional|enums:expiry,price,createdAt',
      sortOrder: 'optional|enums:ascending,descending',
    }

    const { errors } = validate(rules, req.query);
    if(errors) {
      return res.status(400).json({ message: errors[0] });
    }

    return next();
  } catch (error) {
    console.log(errors);
    return res.status(500).json({ message: 'server error' })
  }
}
```

Here we have assumed every fields in request query to be optional and also we have added string check for each field.

You may have noticed that some fields like `productId`, `expiryMin`, `createdAtMax`, etc. don't have any `string` rule. This is because some rules like `mongoid`, `date`, `enums:` automatically checks for string. So you don't need to write `string` check explicitly there.

Guide

1. Separators

`Validator.validate` function requires the `rules` object having key-value pairs where keys are the names of the variables whose needs to be validated. Then the values are the rules in `string` format separated by pipe `|` operator without any spaces. You can give multiple rules for any value. e.g.

```
let rules = {
  name: `string|min:2|max:8`,
};
```

In this example, we are giving 3 rules to the name field:

1. `string`: As the name says, it means, the `name` field must be string
2. `min:2`: The `name` field must be at least 2 characters long
3. `max:8`: The `name` field must be at most 8 characters long

> **Note:** Instead of string validations with `|` separation you can also give validations in arrays. Here's an equivalent example for the same above rule

```
let rules = {
  name: ['string', 'min:2', 'max:8']
}
```

> Also note that array based rules works differently for `$atleast` and `$atmost` keyword. For more details see the API Section of `$atleast` and `$atmost`

2. Data Types

You can add the following data type check:

- `string`
- `number`
- `boolean`
- `array`
- `object`
- `bigint`
- `symbol`

> **NOTE:** By using the combination of `string` and `number` you can check for *numeric string* validation. e.g. `string|natural` will check for *natural number string*. Same works for `string` and `boolean` as well.

3. Automatic String Check

In these cases, it will automatic check for `string` data type, and you don't need to explicitly add a `string` type check:

- `name`
- `fullname`
- `phone`
- `email`
- `url`

- `domain`
- `username`
- `alpha`
- `alphanumeric`
- `mongoid`
- `uuid`
- `date`
- `dateonly`
- `time`
- `lower`
- `upper`
- `ip`

4. Automatic Number Check

In these cases, it will automatic check for `number` data type, and you don't need to explicitly add a `number` type check:

- `int`
- `positive`
- `negative`
- `natural`
- `whole`

5. Optional and Nullable Values

Optional

By default, all the data validation is *compulsory*, if you want any data field to be `optional` (*undefined* or *absent field*), then make sure to add `optional` validation. e.g.

```
let rules = {
  phone: 'optional|phone',
};
```

In this case, `phone` can be optional (means this field can be absent).

Nullable

If you want any field to be `nullable` , then use `nullable` validation, this make sure that the field can be null. e.g.

```
let rules = {
  spouse: 'nullable|name',
};
```

Now, `spouse` can be *null*.

> **NOTE:** In this above case `spouse` can't be *undefined* (its still required). If you want it to be both `optional` and `nullable` then use both:

```
let rules = {
  spouse: 'optional|nullable|name',
};
```

> Now, its not required and even can have null value.

Atleast

Some times it is required to have at least one field among the list of few optional fields. Let's say we have optional `email` and `phone` of a user but atleast one of them should always be present. In that case we can use `$atleast`, this will make sure we have atleast `email` or `phone` field. e.g.

```
let rules = {
  email: 'optional|email',
  phone: 'optional|phone',
  $atleast: 'email|phone'
};
```

Suppose you want at least `n` number of fields to be present from a set of fields. You can achieve this using `size:` keyword. e.g.

```
let rules = {
  name: 'optional|name',
  email: 'optional|email',
  username: 'optional|username',
  phone: 'optional|phone',
  $atleast: 'name|email|username|phone|size:2'
};
```

This example accepts 4 optional fields out of which any 2 is required.

> See the API Section for the validation of more than one group of fields

Atmost

Some times it is required to have at most one field among the list of few optional fields. Let's say we have optional `email` and `phone` of a user but only one of them should be present not both. In that case we can use `$atmost`, this will make sure we have atmost one of `email` or `phone` field. e.g.

```
```

```
let rules = {
  email: 'optional|email',
  phone: 'optional|phone',
  $atmost: 'email|phone'
};
```

Suppose you want at most `n` fields from a set of fields. You can achieve this using `size:` keyword. e.g.

```
let rules = {
  name: 'optional|name',
  email: 'optional|email',
  username: 'optional|username',
  phone: 'optional|phone',
  $atmost: 'name|email|username|phone|size:3'
};
```

This example will accept only 3 fields from 4 optional fields. It will raise error if you try to give more than 3 fields among `name`, `email`, `username`, `phone`

> See the API Section for the validation of more than one group of fields

6. Argument based Validations

It supports the following argument based validations (See API for more details):

- `equal:<value>`: equality check for string, number and boolean
- `size:<int>`: length check for strings and arrays and digits length check for numbers (or numeric strings)
- `min:<value>`: minimum length check for strings and arrays, minimum value check for numbers(or numeric strings) and date(date should be in ISO format)
- `max:<value>`: maximum length check for strings and arrays, maximum value check for numbers(or numeric strings) and date(date should be in ISO format)
- `regex:<regex>`: regular expressions check for strings
- `decimalsize:<value>`: check for exact number of digits after decimal points for numbers and numeric strings
- `decimalmin:<value>`: check for minimum number of digits after decimal points for numbers and numeric strings
- `decimalmax:<value>`: check for maximum number of digits after decimal points for numbers and numeric strings
- `enums:<value>`: check for enum values for strings, numbers and boolean
- `arrayof:<validation>`: check if each elements in the array follows the given `<validation>`. (It has its own separate section)

> **INFO**: for more details and examples, see the API section.

> **LIMITATIONS**: To know about the limitations of `decimalsize:<value>` , `decimalmin:<value>` and `decimalmax:<value>` , see the API section

7. Regular Expression Limitations

If `string` validations has regex which itself has `|` , then it will fail for validation. You can't use pipe operator `|` for regular expressions when using string based validation rules. In those cases, use array based rules instead. e.g.

> **ERROR**: This will not work, as its regex has `|` operator:

```
let rules = {
  gender: 'string|regex:/^(male)|(female)$/',
};
```

> **FIX**: Instead use this:

```
let rules = {
  gender: ['string', 'regex:/^(male)|(female)$/'],
};
```

8. Nested Object Validation

You can also pass object based rule to validate nested object. e.g.

```
const { validate } = require('super-easy-validator');

const rules = {
  address: {
    line1: 'string|min:10',
    line2: 'optional|string|min:10',
    city: 'name',
    state: 'name',
    pin: 'string|natural|size:6',
    country: {
      code: 'alpha|upper|size:2',
      phoneCode: 'regex:/^\\+[0-9]{1,3}$/'
    },
  }
}

const data = {
  address: {
    line1: 800,
    line2: false,
    city: 'N/A',
    state: 'N/A',
    pin: 'ABC123',
    country: {
      phoneCode: '+9999'
    }
```

```
  }
}

let { errors } = validate(rules, data);
if (errors) {
  console.log(errors);
}
```

Output

```
[
  'address.line1 must be string',
  'address.line2 must be string',
  'address.city must be a valid name',
  'address.state must be a valid name',
  'address.pin must be a valid numeric string',
  'address.country.code is required',
  'address.country.phoneCode is invalid'
]
```

There is no limit on the level of nesting.

9. Simple Array Validation

Sometimes it is required to validate each elements in an array. For this purpose we have `arrayof:` validation. It is an argument based validation where the argument itself is a validation rule. e.g.

```
let rules = {
  skills: 'array|min:2|arrayof:string|arrayof:max:10',
};
```

The above example validate the `skills` field only when all these criteria meets:

- `skills` field should be of type *array*
- it should have at least 2 elements
- each element in the `skills` array should be *string*
- each element in the `skills` array should not exceed 10 characters.

Similarly, array of numeric strings can also be checked using this approach:

```
let rules = {
  emails: 'arrayof:string|arrayof:int'
};
```

In the example above, it will check for array where each elements must be integer strings.

Here is one more example for validation of optional emails list (The list could have valid email *string* or *undefined*):

```
let rules = {
  emails: 'arrayof:optional|arrayof:email'
};
```

> **NOTE:** `array` validation is automatically applied when you use `arrayof:` validation rule.

10. Array Objects Validation

You can also validate objects inside an array if you know the objects structure. Here is an example of objects validation which are present inside an array field.

e.g. In this example, we will going to validate each product present in the `products` field:

```
const { validate } = require('./index')

const rules = {
    products: [
        {
            title: 'string|min:5',
            description: 'string|min:20',
            price: 'positive',
            category: 'enums:Electronics,Kitchen,Fashion,Others',
        },
    ],
}

const data = {
    products: [
        {
            title: 'Smartphone',
            description: 'High-performance smartphone with a stunning display and
advanced features.',
            price: 599.99,
        },
        {
            title: 'Coffee Maker',
            price: 'InvalidPrice',
            category: 'Kitchen',
        },
        {
            title: 'Designer Dress',
            description: 'Elegant and stylish designer dress for special
occasions.',
            price: 149.99,
            category: 'InvalidCategory',
        },
        {
```

```
            title: 123,
            description: 'Premium-quality notebook for all your creative and
professional needs.',
            price: '29.99',
            category: 'Others',
        },
        {
            title: 'Invalid',
            description: 'Short desc',
            price: -10,
            category: 'Fashion',
        },
        {
            title: 'Laptop',
            description: 'Powerful laptop with cutting-edge technology.',
            price: 999.99,
            category: 'Electronics',
        },
    ],
}

let { errors } = validate(rules, data)
if (errors) {
    console.log(errors)
}
```

Output

```
[
  'products[0].category is required',
  'products[1].description is required',
  'products[1].price must be a valid number',
  'products[2].category is invalid',
  'products[3].title must be string',
  'products[3].price must be a valid number',
  'products[4].description must have length of at least 20',
  'products[4].price must be a valid positive number'
]
```

There is no limit on the level of nesting. You can also use the combination of nested object and nested array objects validation rules.

11. Error Options

In case of errors, you can change field names to be shown in error messages and you can even set your own custom error message for each fields. You can also set(or hide) the quotes type in error message

- `field:<value>`: Using this you can set custom field name for a specific field. This field name will then be used in error message instead of the original field name. Make sure it is added at the end

> e.g.

```
let rules = {
  age: 'natural|field:person age'
}
```

> This is will show the following error message (for age = 4.5)

```
`person age` must be a valid natural number
```

- `error:<value>`: Using this you can set custom error message for any field. Make sure it is added at the end

> e.g.

```
let rules = {
  age: 'natural|error:given age is not valid'
}
```

> This is will show the following error message (for age = 4.5)

```
given age is not valid
```

- `{quotes: <option>}`: This option can be set in the validate function as 3rd parameter. It will set the quotes with either single quotes, double quotes, backtick or no quotes. Possible options are: `none` (default), `single-quotes`, `double-quotes` and `backtick`.

> e.g.

```
let rules = {
  name: 'name',
  age: 'natural',
};

let data = {
  name: '...',
  age: -7
}

const {errors} = Validator.validate(rules, data, {quotes: 'double-quotes'})

console.log(errors)
```

> This is will show the following error messages:

```
[
  '"name" must be a valid name',
  '"age" must be a valid natural number'
]
```

12. Strict Check

Sometimes you don't want to have any extra fields in the given data except those which are already defined in validation rules. To do this strict check, you can pass a `strict` option to `true` at the time of validation.

e.g. Suppose you want a user object should have 3 fields: `name`, `age` and `gender`. But at the same time you also don't want to have any extra fields in this user object. Here is how you add those checks:

```
const { validate } = require('super-easy-validator')

const rules = {
  name: 'name',
  age: 'natural|min:18',
  gender: 'enums:male,female'
}

const user = {
  name: 'john doe',
  age: 30,
  gender: 'male',
  hobby: 'web development'
}

const { errors } = validate(rules, user, { strict: true })
if(errors) {
  console.log(errors)
}
```

Final Output

```
[ 'hobby is not required' ]
```

> **NOTE:** When strict check is enabled, it strict checks for nested objects and nested array objects as well.

API

Optional and Nullable Types

1. `optional`

`optional` validation makes the variable optional means it can be *undefined* or *missing field*.

```
let rules = {
  organization: 'optional|string',
};
```

In the example above, organization field can be absent or it must be *string*.

2. `nullable`

`nullable` validation makes the variable nullable means it can be *null*.

```
let rules = {
  organization: 'null|string',
};
```

In the example above, organization field can be either *null* or *string*.

> **NOTE:** If you want any field to support both *undefined* or *null*, then apply both validation. e.g.

```
let rules = {
  age: 'optional|null|number',
};
```

> In the example above, the `age` could be absent, *null* or even any number.

3. `$atleast`

Some times it is required to have at least one field among the list of few optional fields. Suppose we have optional `otpUnlock`, `faceUnlock` and `pinUnlock` fields, but atleast one of them is required. In that case we can use `$atleast`. e.g.

```
let rules = {
  otpUnlock: 'optional|boolean',
  faceUnlock: 'optional|boolean',
  pinUnlock: 'optional|boolean',
  $atleast: 'otpUnlock|faceUnlock|pinUnlock'
};
```

This will make sure at least one of the 3 is `true`.

Suppose you want at least `n` number of fields to be present from a set of fields. You can achieve this using `size:` keyword. e.g.

```
let rules = {
  name: 'optional|name',
  email: 'optional|email',
  username: 'optional|username',
  phone: 'optional|phone',
  $atleast: 'name|email|username|phone|size:2'
};
```

This example accepts 4 optional fields out of which any 2 is required.

Now, suppose you need check from a group of fields multiple time. i.e. Lets say there is a situation where any 2 is required from a set of (`name`, `age`, `gender`), not only this any 1 is also required among (`id`, `email`, `username`). Then, you can use this array based validation rule as shown in this example:

```
let rules = {
  name: 'optional|name',
  age: 'optional|natural|min:18',
  gender: 'optional|enums:male,female',
  id: 'optional|uuid',
  email: 'optional|email',
  username: 'optional|username',
  $atleast: ['name|age|gender|size:2', 'id|email|username']
}
```

This validation rule create separate errors for the group of (`name`, `age`, `gender`) and (`id`, `email`, `username`)

4. `$atmost`

Some times it is required to have at most one field among the list of few optional fields. Suppose we have optional `otpUnlock`, `faceUnlock` and `pinUnlock` fields, but atmost one of them is allowed at a time. In that case we can use `$atmost`. e.g.

```
let rules = {
  otpUnlock: 'optional|boolean',
  faceUnlock: 'optional|boolean',
  pinUnlock: 'optional|boolean',
  $atmost: 'otpUnlock|faceUnlock|pinUnlock'
};
```

This will make sure at most one of the 3 fields is given.

Suppose you want at most `n` number of fields to be present from a set of fields. You can achieve this using `size:` keyword. e.g.

```
```

```
let rules = {
  name: 'optional|name',
  email: 'optional|email',
  username: 'optional|username',
  phone: 'optional|phone',
  $atmost: 'name|email|username|phone|size:2'
};
```

This example accepts 4 optional fields out of which only 2 can be given at a time.

Now, suppose you need check from a group of fields multiple time. i.e. Lets say there is a situation where only 2 is required from a set of (`name`, `age`, `gender`), also, only 1 is required among (`id`, `email`, `username`). Then, you can use this array based validation rule as shown in this example:

```
let rules = {
  name: 'optional|name',
  age: 'optional|natural|min:18',
  gender: 'optional|enums:male,female',
  id: 'optional|uuid',
  email: 'optional|email',
  username: 'optional|username',
  $atmost: ['name|age|gender|size:2', 'id|email|username']
}
```

This validation rule create separate errors for the group of (`name`, `age`, `gender`) and (`id`, `email`, `username`)

Data Types

1. `string`

`string` validation is used to check if a field is string or not. e.g.

```
let rules = {
  fieldName: 'string',
};
```

> **Note:** This validation automatically applies in cases of: `email` , `url` , `domain` , `name` , `username` , `alpha` , `alphanumeric` , `phone` , `mongoid` , `date` , `dateonly` , `time` , `lower` , `upper` , `ip` , and `regex:<value>` .

2. `number`

`number` validation is used to check if a field is number or not. e.g.

```
let rules = {
  fieldName: 'number',
};
```

> **Note:** This validation automatically applies in cases of: `int` , `positive` , `negative` , `natural` , `whole` .

3. `boolean`

`boolean` validation is used to check if a field is boolean (*true* or *false*). e.g.

```
let rules = {
  fieldName: 'boolean',
};
```

> **Note:** You can use a combination of `string` and `boolean` to check if a field is boolean string. e.g.

```
let rules = {
  active: 'string|boolean',
};
```

> the above example will check if `active` is one of `'true'` or `'false'`

4. `array`

`array` validation is used to check if a field is an array. e.g.

```
let rules = {
  fieldName: 'array',
};
```

5. `object`

`object` validation is used to check if a field is an *object*. e.g.

```
let rules = {
  address: 'object',
  'address.pin': 'regex:/^[0-9]{6}$/',
  'address.city': 'name',
};
```

You can also write validation for nested properties of object using dot `.`

> **NOTE:** In case if `address` is array, it will fail the validation. Although array in javascript is also an object but it will still fail, as array and object have completely different roles.

6. `bigint`

`bigint` validation is used to check if a field is a *bigint* type. e.g.

```
let rules = {
  field: 'bigint',
};
```

7. `symbol`

`symbol` validation is used to check if a field is a *symbol* type. e.g.

```
let rules = {
  field: 'symbol',
};
```

Specific String Types

> **NOTE:** These validations automatically check if the field is a *string* data type.

1. `email`

`email` validation is used to check if a field is a valid email string. e.g.

```
let rules = {
  myEmail: 'email',
};
```

2. `url`

`url` validation is used to check if a field is a valid URL string. e.g.

```
let rules = {
  profile: 'url',
};
```

3. `domain`

`domain` validation is used to check if a field is a valid domain string. e.g.

```
let rules = {
  domain: 'domain',
};
```

4. `name`

`name` validation is used to check if a field is a valid name (can be full name or short name) string. e.g.

```
let rules = {
  studentName: 'name',
};
```

5. `fullname`

`fullname` validation is used to check if a field is a valid fullname (must be 2 or 3 words separated with spaces) string. e.g.

```
let rules = {
  studentName: 'fullname',
};
```

6. `username`

`username` validation is used to check if a field is a valid username string. This is primarily used to generate a unique ID for each user for any particular site. e.g.

```
let rules = {
  username: 'username',
};
```

7. `alpha`

`alpha` validation is used to check if a string contains only alphabets. This includes lowercase as well as uppercase alphabets. e.g.

```
let rules = {
  keyword: 'alpha',
};
```

8. `alphanumeric`

`alphanumeric` validation is used to check if a string contains only alphabets and digits. This includes lowercase as well as uppercase alphabets. e.g.

```
let rules = {
  passwordHash: 'alphanumeric',
};
```

```
```

9. phone

`phone` validation is used to check if a string contains valid phone number. This could also include a `+` or even a `space` . e.g.

```
let rules = {
  phone: 'optional|phone',
};
```

10. mongoid

`mongoid` validation is used to check if a string is valid mongodb ID. e.g.

```
let rules = {
  userId: 'mongoid',
};
```

11. date

`date` validation is used to check if a string is valid ISO date. This may also include the time as well. e.g.

```
let rules = {
  dob: 'date',
};
```

This will validate the following strings:

```
'1996-01-10';
'1996-01-10T23:50';
'1996-01-10T23:50:34';
'1996-01-10T23:50:34.6';
'1996-01-10T23:50:34.6789';
'1996-01-10T23:50:34.6789Z';
'1996-01-10T23:50:34.6789+05:30';
'1996-01-10T23:50:34.6789-03:00';
```

12. dateonly

`dateonly` validation is used to check if a string is valid ISO date. This must include date only without time. e.g.

```
```

```
let rules = {
  dob: 'dateonly',
};
```

This will validate the following strings:

```
'1996-01-10';
'2023-12-30';
'9999-04-01';
```

13. `time`

`time` validation is used to check if a string is valid ISO time. This must include time only. e.g.

```
let rules = {
  startedAt: 'time',
};
```

This will validate the following time strings:

```
'23:55:00';
'23:55:00.34';
'23:55:00.3400';
```

14. `lower`

`lower` validation is used to check if a string is all lowercase. This could include any character except uppercase letters. e.g.

```
let rules = {
  name: 'lower',
};
```

15. `upper`

`upper` validation is used to check if a string is all uppercase. This could include any character except lowercase letters. e.g.

```
let rules = {
  name: 'upper',
};
```

16. `ip`

`ip` validation is used to check if a string is a valid IP address. e.g.

```
let rules = {
  name: 'ip',
};
```

17. `uuid`

`uuid` validation is used to check if a string is valid UUID (used in SQL Table IDs). e.g.

```
let rules = {
  userId: 'uuid',
};
```

Specific Number Validation

> **NOTE:** These validations automatically check if the field is a *number* data type.

1. `int`

`int` validation is used to check if a number is valid integer. e.g.

```
let rules = {
  temperature: 'int',
};
```

> **NOTE:** This can be used along with `string` to check for integer strings. e.g.

```
let rules = {
  temperature: 'string|int',
};
```

> The above validation will check if `temperature` is a valid integer string.

2. `positive`

`positive` validation is used to check if a number is a positive number (Can't be 0 or negative). e.g.

```
let rules = {
  price: 'positive',
```

```
};
```

> **NOTE:** This can be used along with `string` to check for positive numeric strings. e.g.

```
let rules = {
  price: 'string|positive',
};
```

> The above validation will check if `price` is a valid positive number in string format.

3. `negative`

`negative` validation is used to check if a number is a negative number (Can't be 0 or positive). e.g.

```
let rules = {
  concaveFocalLength: 'negative',
};
```

> **NOTE:** This can be used along with `string` to check for negative numeric strings. e.g.

```
let rules = {
  depth: 'string|negative',
};
```

> The above validation will check if `depth` is a valid negative number in string format.

4. `natural`

`natural` validation is used to check if a number is a valid natural number (must be positive integers). e.g.

```
let rules = {
  iq: 'natural',
};
```

> **NOTE:** This can be used along with `string` to check for natural numeric strings. e.g.

```
let rules = {
  iq: 'string|natural',
};
```

> The above validation will check if `iq` is a valid natural number in string format.

5. `whole`

`whole` validation is used to check if a number is a valid whole number (can be positive integers or 0). e.g.

```
let rules = {
  score: 'whole',
};
```

> **NOTE:** This can be used along with `string` to check for whole number strings. e.g.

```
let rules = {
  score: 'string|whole',
};
```

> The above validation will check if `score` is a valid whole number in string format.

Argument Based Validations

These validations require some argument(s).

1. `equal`

`equal:<value>` validation is used to check if a field has a specific value. It works for string, number and boolean data type

- **For numbers**, it will check if the number equals `<value>`
- **For strings**, it will check if the string equals `<value>`
- **For boolean**, it will check if the boolean equals `<value>`.

```
let rules = {
  marks: 'number|equal:100', // marks should be number with value 100
  isTopper: 'boolean|equal:true', // isTopper should be boolean with value true
  name: 'string|equal:sia', // name should be string with value 'sia'
  status: 'equal:200', // auto detect
};
```

In this case, if the `status` is a string type then it must be `'200'` , else if the status is number, then it must be `200` .

2. `size`

`size:<value>` validation is used to check if a field has a specific size.

- **For numbers**, it will check if the number size is of `<value>` digits.
- **For strings**, it will check if the string length is `<value>`.
- **For arrays**, it will check if the array length is `<value>`.

e.g.

```
let rules = {
  foods: 'array|size:3', // array should have 3 elements
  otp: 'string|natural|size:6', // natural number string should have 6 digits
  pinCode: 'number|size:5', // pinCode should be number and must have 5 digits
  field: 'size:4', // Auto detect
};
```

> **NOTE:** In case if the data type is not defined as in previous case for `field`, then it will detect the data type of the `field` and then apply this validation but only if `field` is of type `string`, `number` or `array`.

3. `min`

`min:<value>` validation is used to check if a field has minimum of value `<value>`.

- **For numbers & numeric strings**, it will check if the number is more than or equal to `<value>`.
- **For strings**, it will check if the string length is more than or equal to `<value>`.
- **For arrays**, it will check if the array length is more than or equal to `<value>`.
- **For date string**, it will check if the date is more than or equal to `<value>`. Make sure that the date should be in ISO string format.

e.g.

```
let rules = {
  foods: 'array|min:3', // array should have minimum 3 elements
  otp: 'string|natural|min:6', // natural number string with minimum value 6
  pinCode: 'number|min:5', // pinCode should be >= 5
  dob: 'date|min:2023-01:01T11:50:34.9876Z', // date should be >= '2023-
01:01T11:50:34.9876Z'
  field: 'min:4', // Auto detect
};
```

> **NOTE:** In case if the data type is not defined as in the previous case for `field`, then it will detect the data type of the `field` and then apply `min:` validation but only if `field` is of type `string`, `number`, `array` or `date`.

4. `max`

`max:<value>` validation is used to check if a field has maximum of value `<value>`.

- **For numbers & numeric strings**, it will check if the number is less than or equal to `<value>`.
- **For strings**, it will check if the string length is less than or equal to `<value>`.
- **For arrays**, it will check if the array length is less than or equal to `<value>`.
- **For date string**, it will check if the date is less than or equal to `<value>`. Make sure that the date should be in ISO string format.

e.g.

```
let rules = {
  foods: 'array|max:3', // array should have maximum 3 elements
  otp: 'string|natural|max:6', // natural number string should with maximum value
6
  pinCode: 'number|max:5', // pinCode should be <= 5
  dob: 'date|max:2023-01:01T11:50:34.9876Z', // date should be <= '2023-
01:01T11:50:34.9876Z'
  field: 'max:4', // Auto detect
};
```

> **NOTE:** In case if the data type is not defined as in the previous case for `field`, then it will detect the data type of the `field` and then apply `max:` validation but only if `field` is of type `string`, `number`, `array` or `date`.

5. regex

`regex:<value>` validation is used to check if string matches a specific regular expression `<value>`. This will automatically apply `string` validation, as it only works for string data type.

e.g.

```
let rules = {
  hash: 'regex:/^[A-Z0-9]{128}$/i', // hash should match this regular expression
};
```

> **ERROR:** In case if the regular expression contains pipe `|` operator, then it will not work as expected as `|` is a special operator used for separation purpose. e.g. The following is incorrect validation:

e.g.

```
let rules = {
  gender: 'string|regex:/^(male)|(female)$/',
};
```

> **FIX:** To get rid of this issue, write validation using array instead of string. e.g.

```
let rules = {
  gender: ['string', 'regex:/^(male)|(female)$/i'],
};
```

> Now, it will work fine.

6. `decimalsize`

`decimalsize:<value>` validation is used to check if a field has `<value>` digits after decimal point.

- **For numbers**, it will check if the number has `<value>` digits after decimal point.
- **For strings**, it will check if the numeric string has `<value>` digits after decimal point. In this case it will automatically apply `string|number` validation.

e.g.

```
let rules = {
  price: 'string|number|decimalsize:2', // price should be a numeric string with 2 digits after decimal point
  pi: 'number|decimalsize:6', // pi should be number with 6 digits after decimal point
  rate: 'decimalsize:3', // Auto detect
};
```

> **NOTE:** In case if the data type is not defined as in the previous case for `rate` , then it will detect the data type of the `rate` and then apply `decimalsize:` validation but only if `rate` is of type `string` or `number` .

> **WARNING:** In case if the data type is `number` , then it will trim the last zeroes after decimal point and then apply validation. So, The following example may not work as expected.

```
let rules = {
  score: 'number|decimalsize:3',
};
```

> This will fail the validation if score is `23.340` or `23.000` because the last zeroes after decimal point will be removed and then it will check for validation. But this only happens with `number` and not with `string|number` string.

7. `decimalmin`

`decimalmin:<value>` validation is used to check if a field has minimum `<value>` digits after decimal point.

- **For numbers**, it will check if the number has minimum `<value>` digits after decimal point.
- **For strings**, it will check if the numeric string has minimum `<value>` digits after decimal point. In this case it will automatically apply `string|number` validation.

e.g.

```
let rules = {
  price: 'string|number|decimalmin:2', // price should be a numeric string with minimum 2 digits after decimal point
  pi: 'number|decimalmin:6', // pi should be number with minimum 6 digits after
```

```
decimal point
  rate: 'decimalmin:3', // Auto detect
};
```

> **NOTE:** In case if the data type is not defined as in the previous case for `rate` , then it will detect the data type of the `rate` and then apply `decimalmin:` validation but only if `rate` is of type `string` or `number` .

> **WARNING:** In case if the data type is `number` , then it will trim the last zeroes after decimal point and then apply validation. So, The following example may not work as expected.

```
let rules = {
  score: 'number|decimalmin:3',
};
```

> This will fail the validation if score is `23.3400` or `23.000` because the last zeroes after decimal point will be removed and then it will check for validation. But this only happens with `number` and not with `string|number` string.

8. `decimalmax`

`decimalmax:<value>` validation is used to check if a field has maximum `<value>` digits after decimal point.

- **For numbers**, it will check if the number has maximum `<value>` digits after decimal point.
- **For strings**, it will check if the numeric string has maximum `<value>` digits after decimal point. In this case it will automatically apply `string|number` validation.

e.g.

```
let rules = {
  price: 'string|number|decimalmax:2', // price should be a numeric string with
maximum 2 digits after decimal point
  pi: 'number|decimalmax:6', // pi should be number with maximum 6 digits after
decimal point
  rate: 'decimalmax:3', // Auto detect
};
```

> **NOTE:** In case if the data type is not defined as in the previous case for `rate` , then it will detect the data type of the `rate` and then apply `decimalmax:` validation but only if `rate` is of type `string` or `number` .

> **WARNING:** In case if the data type is `number` , then it will trim the last zeroes after decimal point and then apply validation. So, The following example may not work as expected.

```
let rules = {
  score: 'number|decimalmax:3',
```

```
};
```

> This will still pass the validation if score is `23.3400` or `23.00000` because the last zeroes after decimal point will be removed and then it will check for validation. But this only happens with `number` and not with `string|number` string.

9. `enums`

`enums:<values>` validation is used to check if a field has a value which lies in comma separated `<values>` . It works for `string` , `number` and `boolean` data type.

- **For numbers**, it will convert each enum `<values>` into *number* and then check if anyone of them matches with field value.
- **For strings**, it will convert each enum `<values>` into *string* and then check if anyone of them matches with field value.
- **For boolean**, it will convert each enum `<values>` into *boolean* and then check if anyone of them matches with field value. This will only check for boolean values (`true`/`false` and not any *truthy* / *falsy* values)

```
let rules = {
  rating: 'number|enums:1,2,3,4,5', // rating should be any one of these numbers:
1, 2, 3, 4, or 5
  status: 'string|enums:pending,success,failed', // status should be string with
anyone of: 'pending', 'success', or 'failed'
  isMarried: 'boolean|enums:true,false', // isMarried should be boolean with value
either true or false
  subject: 'enums:english,maths,science', // auto detect
};
```

> **NOTE:** In case if the data type is not defined as in the previous case for `subject` , then it will detect the data type of the `subject` and then apply `enums:` validation but only if `subject` is of type `string` , `number` or `boolean` .

> **WARNING:** Don't add spaces while writing enum values as it will then include spaces while doing validation. e.g.

```
let rules = {
  grade: 'string|enums: A+, A, B+, B, C, F',
};
```

> This will check if the grade contains anyone of `' A+'` , `' A'` , `' B+'` , `' C'` and `' F'` , instead of checking `'A+'` , `'A'` , `'B+'` , `'C'` and `'F'` . So make sure to put spaces only when required.

10. `arrayof`

Sometimes it is required to validate each elements in an array. For this purpose we have `arrayof:` validation. It is an argument based validation where the argument itself is a validation rule. e.g.

```
let rules = {
  skills: 'array|min:2|arrayof:string|arrayof:max:10',
};
```

The above example validate the `skills` field only when all criteria meets:

- `skills` field is of type *array*
- it should have at least 2 elements
- each element in the `skills` array should be *string*
- each element in the `skills` array should not exceed 10 characters.

Similarly, to add array of numeric strings or boolean strings, you can use the combination of `arrayof:string` and `arrayof:` with other number validation like `number`, `positive`, `negative`, `int`, `whole` and `natural`. e.g.,

```
let rules = {
  ratings: 'arrayof:string|arrayof:natural|arrayof:max:5'
}
```

The above validation makes sure that the ratings is an array of strings where each elements must be any natural number (in string format)

Here is one more example for validation of optional emails list (The list could have valid email *string* or *undefined*):

```
let rules = {
  emails: 'arrayof:optional|arrayof:email'
};
```

> **NOTE:** `array` validation automatically applied when you use `arrayof:` validation rule.

`arrayof`: supports the following validations:

Data Types

- `arrayof:string`
- `arrayof:number`
- `arrayof:boolean`
- `arrayof:array`
- `arrayof:object`
- `arrayof:bigint`
- `arrayof:symbol`

Specific String Types

- `arrayof:email`
- `arrayof:url`
- `arrayof:domain`
- `arrayof:name`
- `arrayof:username`
- `arrayof:alpha`
- `arrayof:alphanumeric`
- `arrayof:phone`
- `arrayof:mongoid`
- `arrayof:date`
- `arrayof:dateonly`
- `arrayof:time`
- `arrayof:lower`
- `arrayof:upper`
- `arrayof:ip`

Specific Number Types

- `arrayof:int`
- `arrayof:positive`
- `arrayof:negative`
- `arrayof:natural`
- `arrayof:whole`

Argument Based Types

- `arrayof:equal:${string}`
- `arrayof:size:${number}`
- `arrayof:min:${number}`
- `arrayof:max:${number}`
- `arrayof:regex:${string}`
- `arrayof:decimalsize:${number}`
- `arrayof:decimalmin:${number}`
- `arrayof:decimalmax:${number}`
- `arrayof:enums:${string}`

> **NOTE:** These validations follows the same rule as shown in their own sections but this time these will work for each elements in the array

Error Options

In case of errors, you can change field names to be shown in error messages and you can even set your own custom error message for each fields. You can also set(or hide) the quotes type in error message

```
field:
```

Using this you can set custom field name for a specific field in error message. It must be added at the end.

> e.g.

```
let rules = {
  age: 'natural|field:person age'
}
```

> This is will show the following error message (for age = 4.5)

```
``person age` must be a valid natural number'
```

error

Using this you can set custom error message for any field. Make sure it is added at the end

> e.g.

```
let rules = {
  age: 'natural|error:given age is not valid'
}
```

> This is will show the following error message (for age = 4.5)

```
'given age is not valid'
```

> **Error:** If your error message contains any pipe | operator, it will not work as expected. In that case, you can use array based validations. e.g.

```
let rules = {
  name: 'name|error:name is incorrect|please add different name'
}
```

> **FIX:** This will not work as its error message has pipe | operator, in that case you can use the example given below:

```
let rules = {
  name: ['name', 'error:name is incorrect | please add different name']
}
```

> Now it will work fine

`{quotes: <value>}`

This option can be set in the validate function as 3rd parameter. It will set the quotes with either single quotes, double quotes, backtick or no quotes. Possible options are: `none` (default), `single-quotes`, `double-quotes` and `backtick`.

> e.g.

```
let rules = {
  name: 'name',
  age: 'natural',
};

let data = {
  name: '...',
  age: -7
}

const {errors} = Validator.validate(rules, data, {quotes: 'double-quotes'})

console.log(errors)
```

> This is will show the following error messages:

```
[
  '"name" must be a valid name',
  '"age" must be a valid natural number'
]
```