Joi tutorial

last modified July 7, 2020

Joi tutorial shows how to validate values in JavaScript with Hapi Joi module.

Hapi Joi

Hapi Joi is an object schema description language and validator for JavaScript objects.

With Hapi Joi, we create blueprints or schemas for JavaScript objects (an object that stores information) to ensure validation of key information.

Hapi is a simple to use configuration-centric framework with built-in support for input validation, caching, authentication, and other essential facilities for building web and services applications.

Hapi Joi install

First, we install the library.

```
$ node -v
v11.5.0
```

We use Node version 11.5.0.

```
$ npm init -y
$ npm i @hapi/js
```

We initiate the project and install Hapi Joi with `nmp i @hapi/joi`.

Joi validate

Validation is performed with the `validate()` function:

```
validate(value, schema, [options], [callback])
```

The `value` is the value being validated and the `schema` is the validation schema.

The `options` are validation options. The `abortEarly` option stops validation on the first error, otherwise returns all the errors found. It defaults to true. The `convert` option attempts to cast values to the required types. It also defaults to true.

The `callback` is the optional synchronous callback method using the signature `function(err, value)`. If the validation failed, the `err` contains the error reason, otherwise it is `null`. The `value` is the value with any type conversions and other modifiers applied.

Joi version

In the first example, we print the version of Hapi Joi.

version.js

```
const Joi = require('@hapi/joi');

console.log(Joi.version)
```

The version of Joi is stored in `Joi.version`.

```
const Joi = require('@hapi/joi');
```

We include the Hapi Joi module.

```
$ node version.js
15.0.3
```

This is a sample output.

Joi synchronous function

In the following example, we perform the validation inside the synchronous function.

sync_fun.js

```
const Joi = require('@hapi/joi');

const schema = Joi.object().keys({

    username: Joi.string().required(),
    email: Joi.string().email().required()
});

let username = 'Roger Brown';
let email = 'roger@example';
let data = { username, email };

Joi.validate(data, schema, (err, value) => {

    if (err) {

        console.log(err.details);

    } else {

        console.log(value);
    }
});
```

We have two values to validate: username and email.

```
const schema = Joi.object().keys({

    username: Joi.string().required(),
    email: Joi.string().email().required()
});
```

This is the validation schema. The username must be a string and it is required. The email must be a valid email address and it is also required.

```
let username = 'Roger Brown';
let email = 'roger@example';

let data = { username, email };
```

This is the data to be validated.

```
Joi.validate(data, schema, (err, value) => {

    if (err) {

        console.log(err.details);

    } else {

        console.log(value);
    }
});
```

The `Joi.validate()` validates the data with the provided schema.

```
$ node sync_fun.js
[ { message: '"email" must be a valid email',
    path: [ 'email' ],
    type: 'string.email',
    context: { value: 'roger@example', key: 'email', label: 'email' } } ]
```

This is the sample output.

Joi returned vals

In the next example, we don't use the synchronous function; we work with the returned values.

ret_vals.js

```
const Joi = require('@hapi/joi');

const schema = Joi.object().keys({

    username: Joi.string().required(),
    born: Joi.date().required()
});

let username = 'Roger Brown';
let born = '1988-12-11';

let data = { username, born };

const { err, value } = Joi.validate(data, schema);

if (err) {
    console.log(err.details);
} else {
    console.log(value);
}
```

The example validates two values: username and date of birth.

```
const { err, value } = Joi.validate(data, schema);
```

Another way of using `validate()` is to get its return values.

```
if (err) {
    console.log(err.details);
} else {
    console.log(value);
}
```

Based on returned values, we either print the error details or the original values.

```
$ node ret_vals.js
{ username: 'Roger Brown', born: 1988-12-11T00:00:00.000Z }
```

We have no errors, so the validated values are printed.

Joi abortEarly

By default, Joi stops validation on first error. If we want to get all errors, we have to set the `abortEarly` option to `true`.

abort_early.js

```
const Joi = require('@hapi/joi');

const schema = Joi.object().keys({

    username: Joi.string().min(2).max(30).required(),
    password: Joi.string().regex(/^[\w]{8,30}$/),
    registered: Joi.number().integer().min(2012).max(2019),
    married: Joi.boolean().required()
});

let username = 'Roger Brown';
let password = 's#cret12';
let registered = '2011';
let married = false;

let data = { username, password, registered, married };
let options = { abortEarly: false };

const { error, value } = Joi.validate(data, schema, options);

if (error) {
    console.log(error.details);
} else {
    console.log(value);
}
```

In the example, we print all errors that have occurred.

```
const schema = Joi.object().keys({

    username: Joi.string().min(2).max(30).required(),
    password: Joi.string().regex(/^[\w]{8,30}$/),
    registered: Joi.number().integer().min(2012).max(2019),
    married: Joi.boolean().required()
});
```

We have four values to validate.

```
let options = { abortEarly: false };

const { error, value } = Joi.validate(data, schema, options);
```

We set the `abortEarly` option to `false`.

```
$ node abort_early.js
[ { message:
        '"password" with value "s#cret12" fails to match the required pattern: /^[\\w]{8,30}
```

```
    path: [ 'password' ],
    type: 'string.regex.base',
    context:
        { name: undefined,
        pattern: /^[\w]{8,30}$/,
        value: 's#cret12',
        key: 'password',
        label: 'password' } },
    { message: '"registered" must be larger than or equal to 2012',
    path: [ 'registered' ],
    type: 'number.min',
    context:
        { limit: 2012,
        value: 2011,
        key: 'registered',
        label: 'registered' } } ]
```

We have two validation errors.

Joi casting values

Joi casts values by default; to disable casting, we set `convert` option to `false`.

casting.js

```
const Joi = require('@hapi/joi');

const schema = Joi.object().keys({

    timestamp: Joi.date().timestamp(),
    val: Joi.number()
});

let val = '23543';
let timestamp = 1559761841;

let data = { val, timestamp };

const { error, value } = Joi.validate(data, schema);

if (error) {
    console.log(error.details);
} else {
    console.log(value);
}
```

In the example, we have two values automatically casted by Joi.

```
let val = '23543';
let timestamp = 1559761841;
```

The string is casted to a number and the timestamp to an ISO string.

```
$ node casting.js
{ val: 23543, timestamp: 1970-01-19T01:16:01.841Z }
```

This is the output.

Joi validate numbers

With `Joi.number()` we can validate numbers.

numbers.js

```
const Joi = require('@hapi/joi');

const schema = Joi.object().keys({

    age: Joi.number().min(18).max(129),
    price: Joi.number().positive(),
    experience: Joi.number().greater(5)
});

let age = 35;
let price = -124.3;
let experience = 6;

let data = { age, price, experience };

const { error, value } = Joi.validate(data, schema);

if (error) {
    console.log(error.details);
} else {
    console.log(value);
}
```

In the example, we validate three numbers.

```
const schema = Joi.object().keys({

    age: Joi.number().min(18).max(129),
    price: Joi.number().positive(),
    experience: Joi.number().greater(5)
});
```

The `age` value must be a number between 18-129. The `price` must be positive and the `experience` must be greater than five.

```
$ node numbers.js
[ { message: '"price" must be a positive number',
    path: [ 'price' ],
    type: 'number.positive',
    context: { value: -124.3, key: 'price', label: 'price' } } ]
```

Since the price was negative, we get this error details.

Joi validate dates

With `Joi.date()`, we can validate dates.

dates.js

```
const Joi = require('@hapi/joi');

const schema = Joi.object().keys({

    timestamp: Joi.date().timestamp(),
    isodate: Joi.date().iso(),
    registered: Joi.date().greater('2018-01-01')
});

let timestamp = 1559761841;
let isodate = '1970-01-19T01:16:01.841Z';
let registered = '2019-02-12';

let data = { timestamp, isodate, registered };

const { error, value } = Joi.validate(data, schema);

if (error) {
    console.log(error.details);
} else {
    console.log(value);
}
```

The example validates three date values.

```
const schema = Joi.object().keys({

    timestamp: Joi.date().timestamp(),
    isodate: Joi.date().iso(),
    registered: Joi.date().greater('2018-01-01')
});
```

In the schema, we have rules to validate that a value is a timestamp, has ISO format, and is greater that a specified value.

```
$ node dates.js
{ timestamp: 1970-01-19T01:16:01.841Z,
    isodate: 1970-01-19T01:16:01.841Z,
    registered: 2019-02-12T00:00:00.000Z }
```

All values were validated OK. Note that the values were automatically casted to the ISO format.

Joi validate arrays

The arrays can be validate with `array()`.

arrays.js

```
const Joi = require('@hapi/joi');

const schema = Joi.array().min(2).max(10);

let data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ];

const { error, value } = Joi.validate(data, schema);

if (error) {
    console.log(error.details);
} else {
    console.log(value);
}
```

In the example, we validate an array of integers.

```
const schema = Joi.array().min(2).max(10);
```

We validate that our array has at least two elements and at max ten elements.

```
$ node arrays.js
[ { message: '"value" must contain less than or equal to 10 items',
    path: [],
    type: 'array.max',
    context:
        { limit: 10, value: [Array], key: undefined, label: 'value' } } ]
```

This is the output.

Joi validate functions

Functions can be validated with `func()`.

functions.js

```
const Joi = require('@hapi/joi');

const schema = Joi.func().arity(2);

function add2int(x, y) {

    return x + y;
}

const { error, value } = Joi.validate(add2int, schema);
```

```
if (error) {
    console.log(error.details);
} else {
    console.log(value);
}
```

In the example, we validate a function.

```
const schema = Joi.func().arity(2);
```

We validate the function's arity (the number of parameters).

In this tutorial, we have done validation in JavaScript with Hapi Joi module.

List all JavaScript tutorials.

Home Facebook Twitter Github Subscribe Privacy