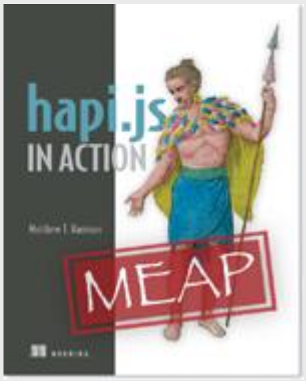


hapi.js in Action: Introducing Joi

By Matt Harrison

In this article, I'll talk about Joi, a Node module for data validation that can validate any kind of data from simple scalar data types such as strings, numbers or booleans, or complex values consisting of several levels of nested objects and arrays.

Humans often make mistakes, and as a result the systems we create need to be prepared for misuse. Validation is an essential part of almost every system we use in our daily lives. Let's use a snack vending machine to illustrate such a system.

A vending machine has several inputs that it needs to validate. If any of the inputs don't match its expectations, the machine will halt normal functioning and give some feedback to the user on what went wrong. For instance, if you place a foreign coin in the slot, the machine will reject the coin and spit it out into the coin return tray.

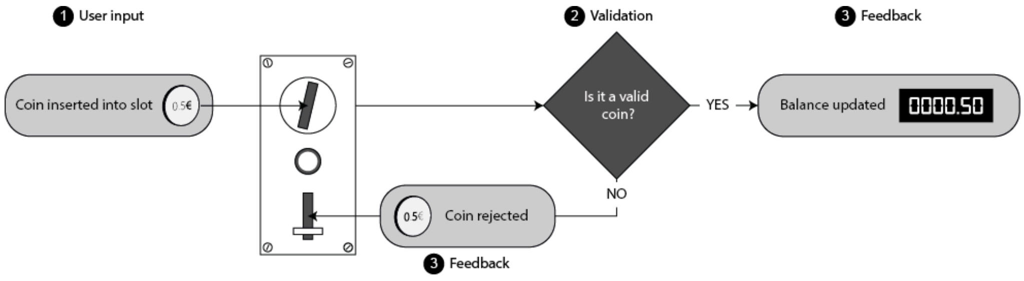


Figure 1 An example of validation that occurs in vending machines

We rely on the feedback we get from validation to make sure we can use systems the correct way. Without it, we'd be clueless as to what we're doing wrong.

We use the term *validation* to apply to software as well. In this article, I'll talk about Joi, a Node nodule for data validation.

Introducing Joi

Joi is a Node module for data validation. Joi can validate any kind of data from simple scalar data types such as strings, numbers or booleans, to complex values consisting of several levels of nested objects and arrays.

(NODE MODULE) JOI (HTTPS://GITHUB.COM/HAPIJS/JOI)

If you're working with some data in your application that comes from an unknown source, for example via a public API, Joi can help you to ensure that data is in the required format. Checking these kind of inputs and acting accordingly when they're incorrect will help to make your applications more stable and reliable.

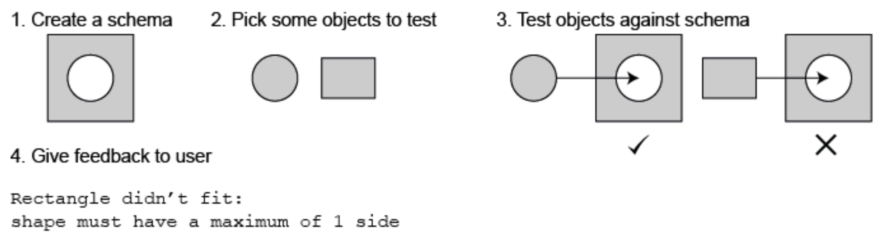
Joi can be used as a standalone module in any Node application. Joi is actually used internally in a lot of the modules in the hapi ecosystem for this exact purpose. If you've used any of hapi's APIs incorrectly before and seen an error message, that message could likely have come from validation performed by Joi.

hapi has been designed with Joi in mind. The framework's built-in validation features, such as validation of HTTP headers and payloads, are designed to work seamlessly with Joi.

How it works

Working with Joi involves a process of four steps. Those steps are shown in figure 2. The first step is to create a *schema*. A schema is an object that describes your expectations and is what you'll be checking your real data against. You can think of a schema as a description of your ideal object.

There's a very common puzzle game given to babies, where they need to place different shaped blocks into different shaped holes. This game is a good analogy to using Joi. With Joi, the schema is like the hole and the object you wish to test is like the block. Any given object is either going to fit the schema, or not.



```
Rectangle didn't fit:
shape must have a maximum of 1 side
```

Figure 2 The steps taken when validating with Joi

Once you have a schema, you will use Joi to test the schema against some objects. If there was a validation error, Joi will tell you exactly what went wrong. You can then use that feedback to make a decision and respond to the user appropriately.

A simple example: Validating a scalar type

A piece of data that we commonly need to validate in our applications is passwords. In this simple example, I will check the format of the `password` argument in an `updatePassword` function. If the format doesn't match the required format, an error will be thrown. The password is required to be a string and be between six and ten characters in length.

Here's the schema for the password validation:

```
var Joi = require('joi');

var schema = Joi.string().min(6).max(10);                        //#A
```

#A A string between 6 and 10 characters in length

FLUENT INTERFACES

Fluent interfaces are an approach to API design in software development. They're also commonly known as chainable interfaces, this is because they consist of methods that are chained onto one another.

Fluent interfaces can promote more readable code where a number of steps are involved and you're not interested in the intermediate returned values.

An example of a fluent interface to make toast could be:

```
var toast = new Toast()
    .cook('3 minutes')
    .spread('butter')
    .spread('raspberry jam')
    .serve();
```

The result of each method call is another `Toast` object but we only care about that delicious final product, which is saved in the `toast` variable, after all methods in the chain have executed.

Joi schemas are also built using a fluent interface. Here's an example of creating a schema for a Javascript date that falls within the month of December 2015, and is formatted in ISO date format:

```
var schema = Joi.date()
    .min('12-1-2015')
    .max('12-31-2015')
    .iso();
```

To test a schema against a real value, you can use `Joi.assert(value, schema)`. When using this function, Joi will throw an error upon encountering the first validation failure. The error message logged will contain some useful information about where the validation failed.

```
var Joi = require('joi');

var schema = Joi.string().min(6).max(10);

var updatePassword = function (password) {

    Joi.assert(password, schema);                                //#A
    console.log('Validation success!');                          //#B
};

updatePassword('password');                                      //#C
updatePassword('pass');                                          //#D
```

#A An error will be thrown here if validation fails
#B If we got here, the validation was successful
#B Valid password (8 characters)
#C Invalid password (4 characters)

If you run this small example in the console you will see some output that resembles the following:

```
Validation success!                                              //#A

index.js:121                                                     //#B
            throw new Error(message + error.annotate());         //#B
                  ^                                              //#B
Error: "pass"                                                    //#B

[1] value length must be at least 6 characters long              //#C
      ...
```

#A Success message after validating value `'password'`
#B Error thrown with with value `'password'`
#C Details about what went wrong

The previous example shows how to validate a simple scalar value, but Joi is capable of much more. Often the values you wish to validate are more complex, compound value such as objects and arrays.

A more complex example: Validating a compound type

Let's imagine that I'm creating an API that collects data from automated weather measuring stations around the world. This data is then persisted and can be retrieved by consumers of the API to get up-to-the-minute data for their region.

Each weather report that is sent by the stations has to follow a standard format. The reports are composed of several fields and can be represented as a JavaScript object. A weather report from Taipei, at noon on a day in July, might look something like the following:

Listing 1 A sample weather report

```
var report = {
    station: 'Taipei',
    datetime: new Date('Wed Jul 22 2015 12:00:00 GMT+0000 (GMT)'),
    temp: 93,
    humidity: 95,
    precipitation: false,
    windDirection: 'E'
};
```

The remote weather stations that are sending the data to the central API are out of my control. It would therefore be prudent to validate all the incoming data to ensure that it matches the standard format. Accepting invalid data from a malfunctioning station could cause unknown problems for consumers of my API.

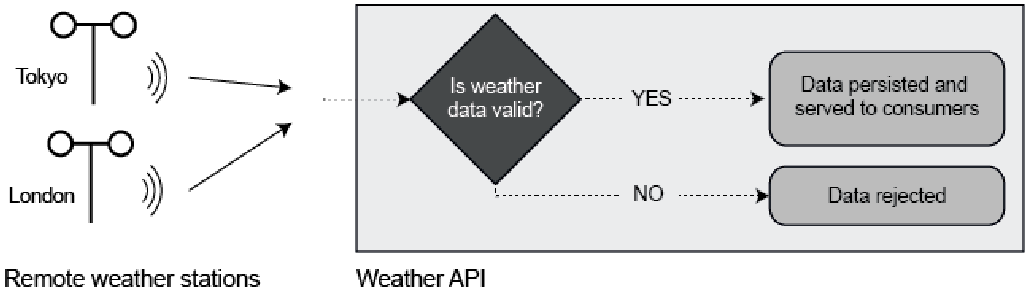


Figure 3 Validating incoming requests helps to ensure the integrity of your data

UNDERSTAND YOUR DATA

The first step is to think about the restrictions I want to put on each field in the report. I've outlined my requirements for the weather reports in table 1.

Table 1 Required format for weather reports

Field name	Datatype	Required	Other restrictions
`station`	String	Yes	Max 100 characters
`datetime`	Date	Yes	

`temp(°F)`	Number	Yes	Between -140 and 140
`humidity`	Number	Yes	Between 0 and 100
`precipitation`	Boolean	No	
`windDirection`	String	No	One of N, NE, E, SE, S, SW, W, NW

CREATE A SCHEMA

In the previous subsection, you saw that we could specify a JavaScript string in a schema using `Joi.string()`. It won't come as a huge surprise to you that the equivalent for an object is `Joi.object()`. Just like `Joi.string()`, `Joi.Object` also has methods you can *chain* onto it to further expand the schema. One of those methods is `.keys()`, this allows you to then add sub-schemas for each of the object's properties. Here's an example of creating a schema for an `object` with keys:

```
var schema = Joi.object().keys({         //#A
      prop1: Joi.string(),               //#B
      ...
});
```

#A Top level schema validates an object
#B Property `prop1` validates a string

Creating schemas for objects with keys is so common that Joi also offers a convenient shorthand for the above. You can replace `Joi.object().keys({...})`, with simply, `{...}`:

```
var schema = {                           //#A
      prop1: Joi.string(),               //#B
      ...
};
```

#A Top level schema validates an object
#B Property `prop1` validates a string

In the listing beneath is the schema for validating the weather reports.

Listing 2 A schema for validating weather reports

```
var schema = {
    station: Joi.string().max(100).required(),               //#A
    datetime: Joi.date().required(),                         //#B
    temp: Joi.number().min(-140).max(140).required(),        //#C
    humidity: Joi.number().min(0).max(100).required(),       //#D
    precipitation: Joi.boolean(),
        //#E
    windDirection: Joi.string()
```

```
        .valid(['N', 'NE', 'E', 'SE', 'S', 'SW', 'W', 'NW'])    //#F
};
```

#A A required `string` **of max 100 character length**
#B A required `date`
#C A required `number` **between -140 and 140**
#D A required `number` **between 0 and 100**
#E An optional `boolean`
#F An optional `string` **with a whitelist of values**

This schema can now be used to validate some real weather reports. You can run the following script to check the sample report from listing 5.1 against this schema

Listing 3 Script to test the report against the schema

```
var Joi = require('joi');

var report = {
    ...
};

var schema = {
    ...
};

Joi.assert(report, schema);
```

There should be no output before the script finishes. That indicates that the value tested was valid. Now let's try to modify the original report so it's no longer valid according to the schema.

Listing 4 Invalid sample weather report

```
var report = {
    station: 'Taipei',
    datetime: new Date('Wed Jul 22 2015 12:00:00 GMT+0000 (GMT)'),
    temp: 34,
    humidity: 93,
    precipitation: false,
    windDirection: 'WE'         //#A
};
```

#A Oops, there's no such direction as WE

If you run the script again, with the modified report, you should see an error thrown. The error message will contain some helpful output to pinpoint exactly what caused the validation to fail:

```
…
Error: {
  "station": "Taipei",
  "datetime": "2015-07-22T12:00:00.000Z",
  "temp": 93,
  "humidity": 95,
  "precipitation": false,
  "windDirection" [1]: "WE"
}

[1] windDirection must be one of N, NE, E, SE, S, SW, W, NW
```

You should now have a pretty good idea of the basics of working with Joi and you should recognize the usefulness of employing validation techniques in your apps. For more information about working with hapi.js, check out my book, hapi.js in Action, from Manning Publications.