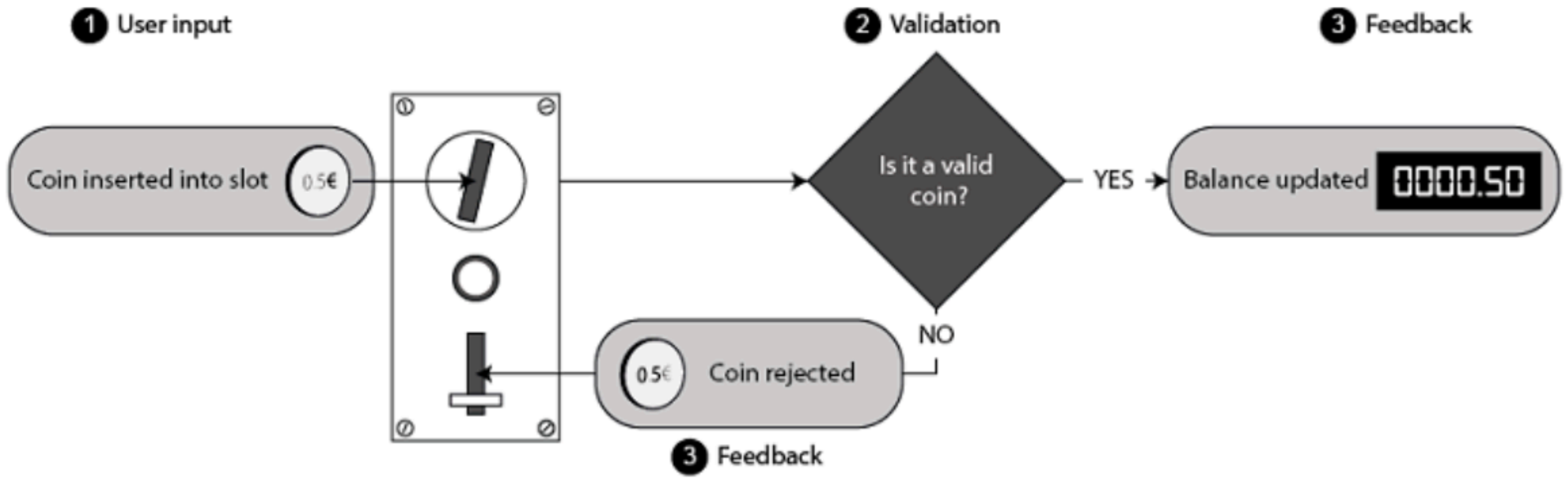
Joi Validation

Validation

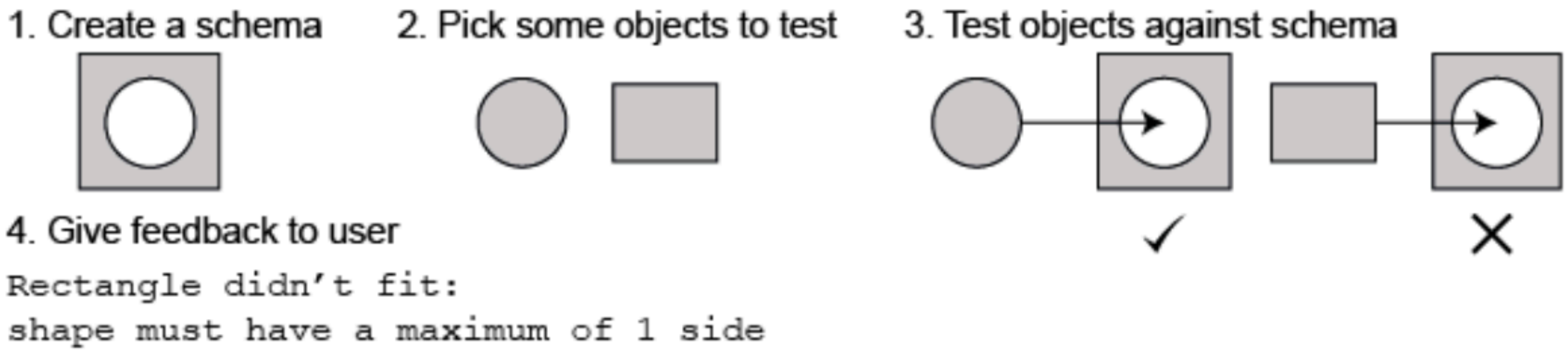


- A vending machine has several inputs that it needs to validate.
- If any of the inputs don’t match its expectations, the machine will halt normal functioning and give some feedback to the user on what went wrong.
- For instance, if you place a foreign coin in the slot, the machine will reject the coin and spit it out into the coin return tray.
- We rely on the feedback we get from validation to make sure we can use systems the correct way

https://github.com/hapijs/joi

- Joi is a Node.js module for data validation.
- Joi can validate any kind of JavaScript values from simple scalar data type such as a string, number or boolean, to complex values consisting of several levels of nested objects and arrays.
- Joi can be used as a standalone module in any Node application.
- hapi has been designed with Joi in mind (rather than the other way around)

How it works: 4 Steps



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

A schema is an object that describes your expectations and is what you’ll be checking your real data against.

Fluent Interfaces

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

- Fluent interfaces are an approach to API design.
- They're also commonly known as chainable interfaces - 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.

- If the return value of each method call is another Toast object…

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

fluent

Fluent Joi Interface

- Joi schemas are also built using a fluent interface.
- A schema for a Javascript date that falls within the month of December 2015, and is formatted in ISO date format

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

Joi Example 1

- 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.

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

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

const updatePassword = function (password)
{
  Joi.assert(password, schema);
  console.log('Validation success!');
};

updatePassword('password');
```

```
Validation success!
```

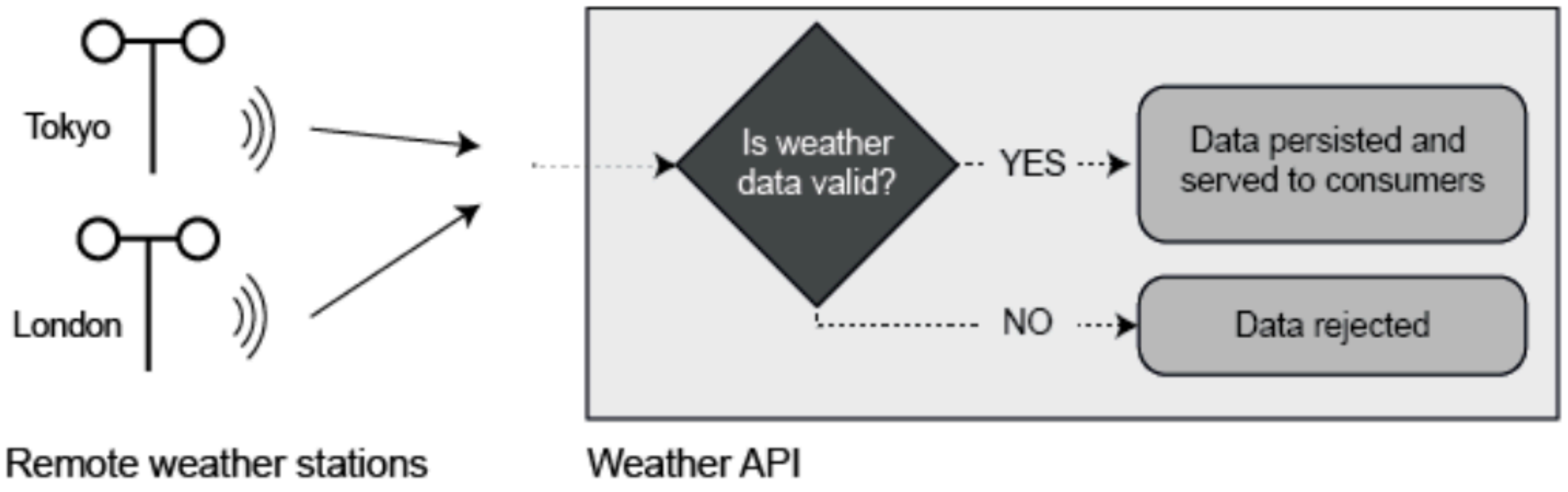
Joi Example 1

- The error message logged will contain some useful information about where the validation failed.

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

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

const updatePassword = function (password)
{
  Joi.assert(password, schema);
  console.log('Validation success!');
};

updatePassword('pass');
```

```
ValidationError: "value" length must be at least 6 characters long
```

Joi Example 2: Scenario



- 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

Joi Example 2: Sample

sample report

```
const report = {
  station: 'Tramore',
  datetime: 'Wed Jul 22 2016 12:00:00 GMT+0800',
  temp: 93,
  humidity: 95,
  precipitation: false,
  windDirection: 'E',
};
```

- Need 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

Joi Example 2: Validation Rules

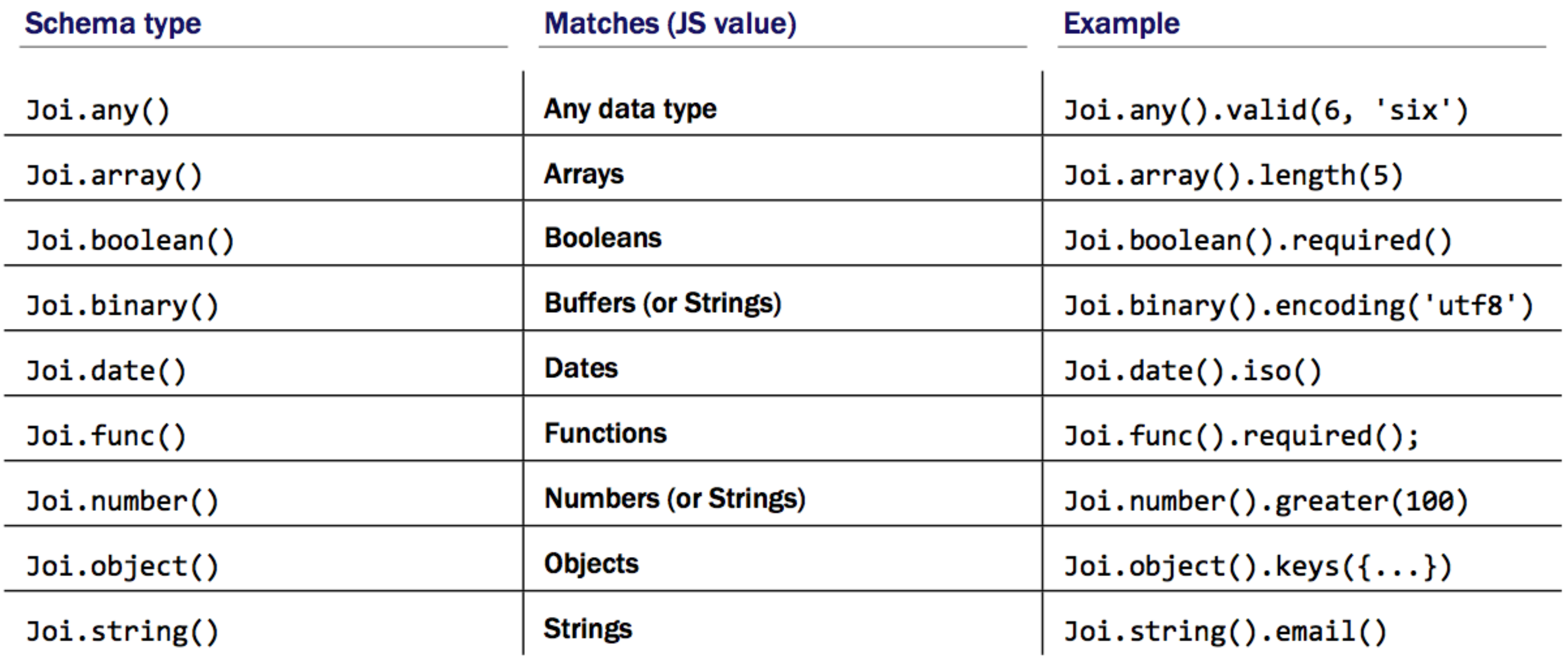
Field name	Datatype	Required	Other restrictions
`station`	`String`	Yes	Max 100 characters
`datetime`	`Date`	Yes	
`temp(ºF)`	`Number`	Yes	Between -140 and 140
`humidity`	`Number`	Yes	Between 0 and 100
`precipitation`	`Boolean`	No	
`windDirection`	`String`	No	One of N, NE, E, SE, S, SW, W, NW

Joi Example 2: Joi Schema

Field name	Datatype	Required	Other restrictions
station	String	Yes	Max 100 characters
datetime	Date	Yes	
temp(ºF)	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

```
const schema = {
  station: Joi.string().max(100).required(),
  datetime: Joi.date().required(),
  temp: Joi.number().min(140).max(140).required(),
  humidity: Joi.number().min(0).max(100).required(),
  precipitation: Joi.boolean(),
  windDirection: Joi.string()
      .valid(['N', 'NE', 'E', 'SE', 'S', 'SW', 'W', 'NW']),
};
```

Joi Schema Types



Schema type	Matches (JS value)	Example
`Joi.any()`	**Any data type**	`Joi.any().valid(6, 'six')`
`Joi.array()`	**Arrays**	`Joi.array().length(5)`
`Joi.boolean()`	**Booleans**	`Joi.boolean().required()`
`Joi.binary()`	**Buffers (or Strings)**	`Joi.binary().encoding('utf8')`
`Joi.date()`	**Dates**	`Joi.date().iso()`
`Joi.func()`	**Functions**	`Joi.func().required();`
`Joi.number()`	**Numbers (or Strings)**	`Joi.number().greater(100)`
`Joi.object()`	**Objects**	`Joi.object().keys({...})`
`Joi.string()`	**Strings**	`Joi.string().email()`

Joi **assert** vs validate

Exception here

This statement never executed

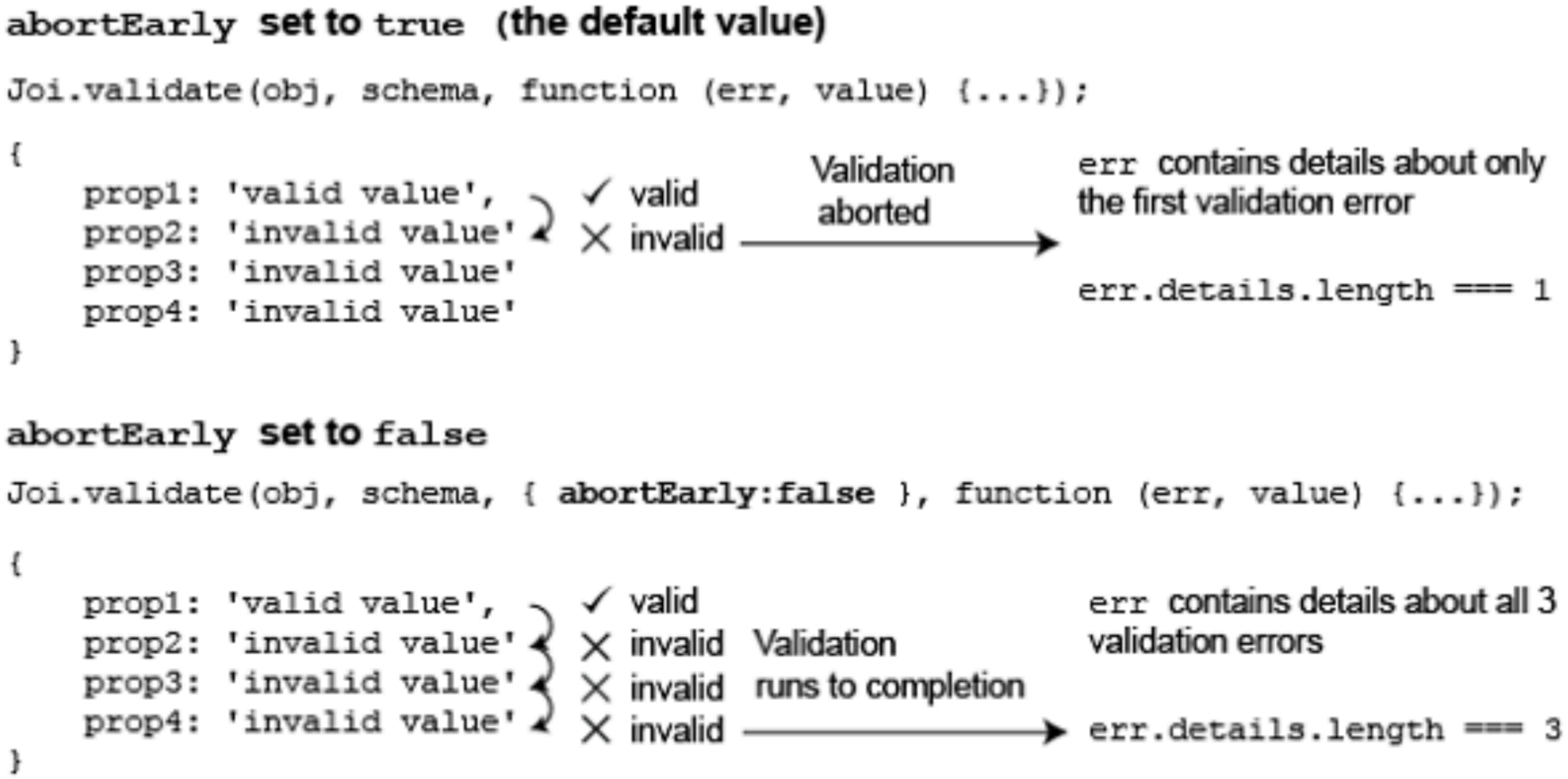
```
const Joi = require('joi');

const fruits = ['mango', 'apple', 'potato'];
const schema = Joi.array().items(['mango', 'apple', 'grape']);

Joi.assert(fruits, schema);

console.log('This code will never execute');
```

Joi assert vs **validate**

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

const fruits = ['mango', 'apple', 'potato'];
const schema = Joi.array().items(['mango', 'apple', 'grape']);

Joi.validate(fruits, schema, (err, value) => {
  if (!err) {
    console.log('The object was valid');
  } else {
    console.log('The object wasn\'t valid');
  }

  console.log('This code will still run');
});
```

- Joi.validate() won’t cause an exception in the program if the tested object doesn’t pass the validation,
- instead it will provide an error object which contains the details of what happened during validation

abortEarly Option



`abortEarly` set to `true` (the default value)

```
Joi.validate(obj, schema, function (err, value) {...});

{
    prop1: 'valid value',
    prop2: 'invalid value'
    prop3: 'invalid value'
    prop4: 'invalid value'
}
```

`err` contains details about only the first validation error

`err.details.length === 1`

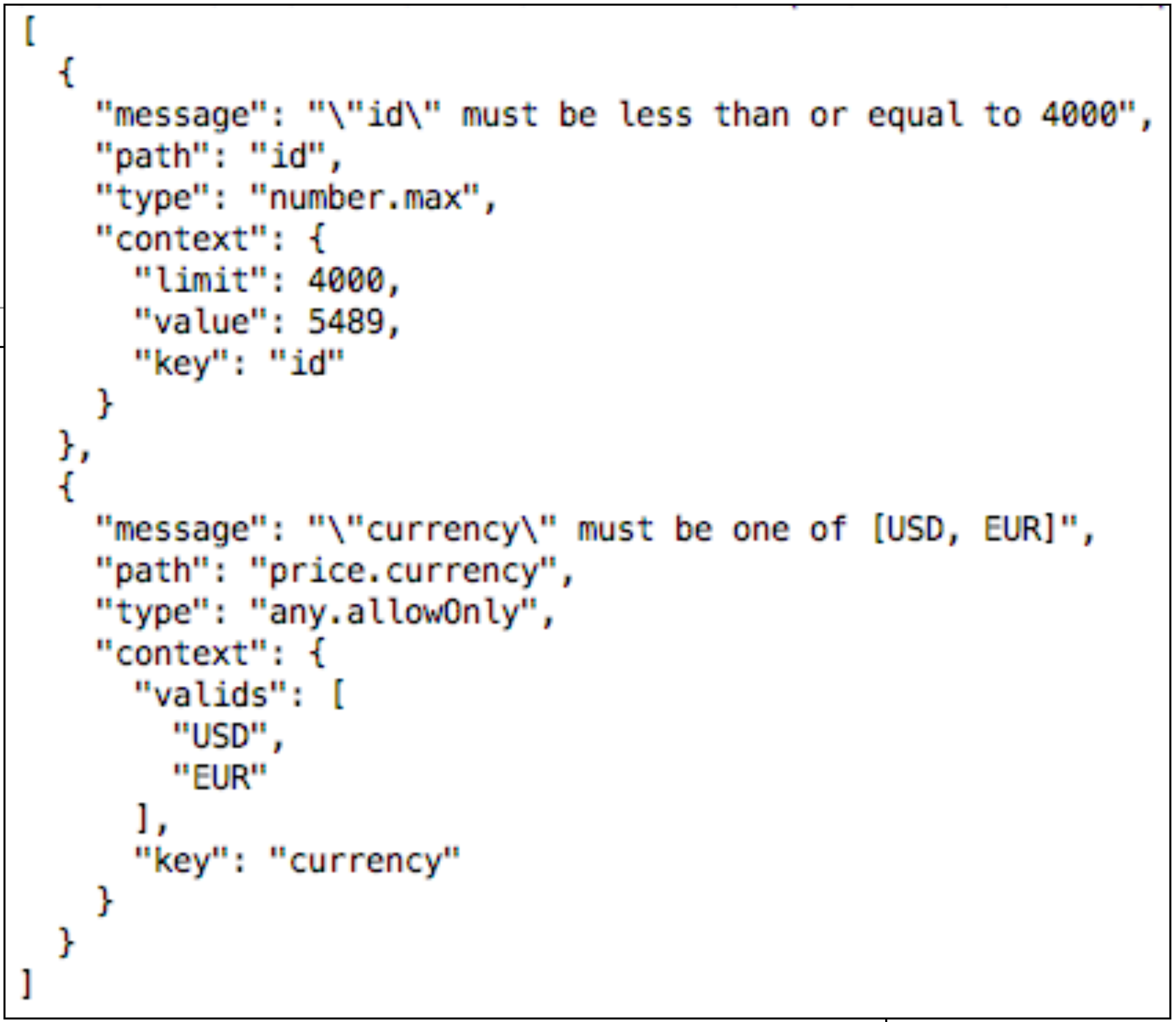
`abortEarly` set to `false`

```
Joi.validate(obj, schema, { abortEarly:false }, function (err, value) {...});

{
    prop1: 'valid value',
    prop2: 'invalid value'
    prop3: 'invalid value'
    prop4: 'invalid value'
}
```

`err` contains details about all 3 validation errors

`err.details.length === 3`

```javascript
const Joi = require('joi');

const product = {
  id: 5489,
  name: 'Trouser press',
  price: {
    value: 34.88,
    currency: 'GBP'
  }
};

const schema = {
  id: Joi.number().max(4000),
  name: Joi.string(),
  price: {
    value: Joi.number(),
    currency: Joi.string().valid(['USD', 'EUR'])
  }
};


Joi.validate(product, schema, { abortEarly: false }, (err, data) => {
  console.log(JSON.stringify(err.details, null, 2));
});
```



```json
[
  {
    "message": "\"id\" must be less than or equal to 4000",
    "path": "id",
    "type": "number.max",
    "context": {
      "limit": 4000,
      "value": 5489,
      "key": "id"
    }
  },
  {
    "message": "\"currency\" must be one of [USD, EUR]",
    "path": "price.currency",
    "type": "any.allowOnly",
    "context": {
      "valids": [
        "USD",
        "EUR"
      ],
      "key": "currency"
    }
  }
]
```