Documentation

Validation library

1. Introduction

This document provides the description of the app for Visitors management and contains next paragraphs:

- Purpose – description of how library could be used.
- Document conventions – data that is needed to understand a description of the library properly (list of definitions, some notes etc).
- References – some useful links.
- Product features – general description of library functionality.
- Operating environment – requirements to the hardware.
- Design and implementation constraints – general description of architecture, list of used programming languages and technologies (frameworks, libraries etc).
- System features – detailed description of each library component.

1.1. Purpose:

Usually it`s necessary to check some data passed into the application. In JavaScript, where Objects are usually used, it`s often required to check the data written in this Object.

The described library was created to provide a simple validation mechanism for Objects in Node JS. It could be simply configured due to your requirements.

The library was initially created as a part of simple Node JS project, but it this library could be easily used separately in another projects.

1.2. Document conventions:

Validation is the process when it`s being checked whether some data is correct and has acceptable format.

1.3. References:

Code of project: https://github.com/vladstudennikov/Visitors-Tracker.

2. Overall description:

2.1. Product features:

The library provides basic architecture for validation layer of some application written on Node JS. It provides you an ability to easily add your own validation rules and validate objects data.

This library is not a ready-made software for validation, but it`s rather an architecture that can help you to build your own validation component for your Node JS project.

2.2. Operating environment:

Hardware requirements: RAM > 1Gb, permanent memory > 50 Mb.

OS – Windows 10, 11.

2.3. Design and implementation constraints:

Stack of technologies:

- JavaScript;
- Node JS (20.11.1);

List of Node JS libraries:

No extra libraries used.

3. Validation process:

An object in JavaScript is a set of fields; each field has its name and value. Value could be represented as a number, character, string or another object.

To validate any field of the object it should be created a class that`s inherited from FieldValidator (in file FieldValidatior.js). The class for field validation should contain the method validate() that receives the value of a field as a parameter. These classes could be placed in file ValidatorClasses.js where all classes for validation are expected to be created, but actually they could be situated anywhere else.

When the class for validation was created, it should be added into validation settings in file ValidationSetup.js. Here enter the name of the Object and then enter names of fields and the classes that contain the function for validating this particular field (this step and ell other steps would be shown below).

Full validation of any object could be made with function validate(type, obj) that is written in file validate.js. This function receives 2 arguments: type – the name of the object; obj – the object itself. This functions returns true if validation was fully passed and false if 1 or more fields of the object are incorrect. Also it could throw 3 exceptions:

- InvalidValildationSetup – invalid settings;
- InvalidFieldNameException – thrown when the field that should be validated was not found in mentioned Object.
- ValidationFunctionWasNotFoundException – thrown when class for validation does not contain valid function validate().

Below you can see an instruction that describes how to setup the validation:

- Let we have an object Visitor that contains next fields: name, surname, patronym, city, address and passport. Let passport is also an object that contains number, issued_by and issued_on fields.

The example of object we would work with:

```
{
  "id": 2,
  "name": "Alice",
  "surname": "Johnson",
  "patronym": "Mary",
  "city": "Los Angeles",
  "address": "456 Elm St",
  "passport": {
    "number": 987654321,
    "issued_by": "California DMV",
    "issued_on": "2022-09-15"
  }
}
```


- Let we want to validate the number of passport – it shoud be an integer positive number. Create the class for validation in file ValidatorClasses.js, it should contain 1 function – validate – that receives 1 parameter – the value of validated field. The class should be inherited from FieldVallidator, and the function validate() should be asynchronous.

Example of the code for validating the passport number:

```
class PassportNumberValidator extends FieldValidator{
    async validate(value) {
        return (Number.isInteger(parseInt(value)));
    }
}
```

- Add validator class into module.exports in file ValidatorClasses.js:

```
module.exports = {
    PassportNumberValidator,

};
```

- Add the class into ValidationSetup.js:

```
module.exports = {
    "Visitor": {
        "passport.number": validation.PassportNumberValidator,
    }
};
```

File ValidationSetup.js contains objects names and settings for its validation. Mention names of fields and validation classes for their validation. Note that

to validate fields in nested objects, type the name of an object, then ".", then – name of field in nested object needed to be validated. In our case passport is nested object, and if we want to validate its field named "number", then we should name this field as "passport.name".

You can add another validation rules:

```
module.exports = {
    "Visitor": {
        "name": validation.NameValidator,
        "surname": validation.SurnameValidator,
        "passport.number": validation.PassportNumberValidator,
        "passport": validation.PassportValidation
    }
};
```

You can set the rules of validation of different object, for example, lets add the validation rules for object named "Car":

```
module.exports = {
    "Visitor": {
        "name": validation.NameValidator,
        "surname": validation.SurnameValidator,
        "passport.number": validation.PassportNumberValidator,
        "passport": validation.PassportValidation
    },

    "Car": {
        "name": validation.CarNameValidator,
        "max_speed": validation.CarMaxSpeedValidator
    }
};
```

- To validate the object call the function validate():

```
await validate("Visitor", toAdd)
```

Function validate() would check all fields in object that should be validated. Note that this function is async.