Schema validation in Node.js using Joi

Securing the application by validating data at the query level with the Joi validation library is very simple. Here is a detailed guide on how to validate schema in Node.js using Joi.

Securing the application by validating data at the query level with the Joi validation library is very simple. Here is **a detailed guide on how to validate schema in Node.js using Joi** .

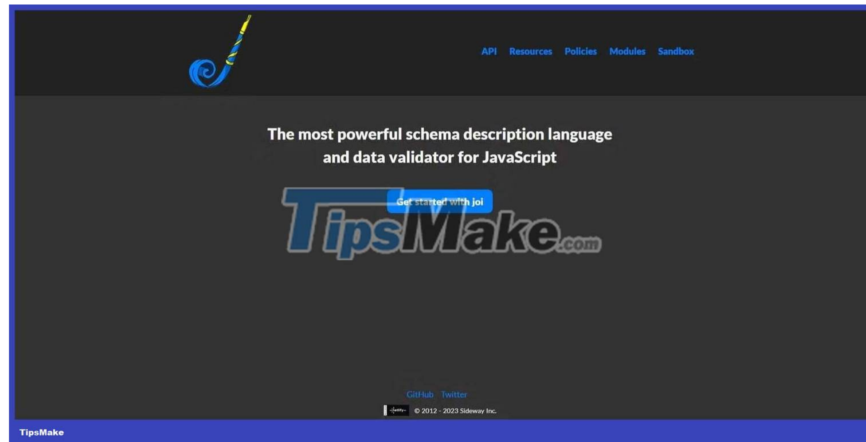
Accepting untested and authenticated data into a web application can create security holes, and unpredictable crashes can arise from invalid data.

Node.js ORMs, such as Sequelize and TypeORM, allow you to set validation rules instantly at the application level. During API development, data travels from HTTP queries to specific endpoints. This happens at the query level, so the default authentication provided by ORMs is not applied to them.

Joi is a schema descriptor and data validation for JavaScript. Here, you will learn how to use the Joi validation library to validate data at the query level.

Set up test project

To demonstrate how Joi validates data, you will build a simple demo application that simulates an actual app.



First, create a project directory and access it by running the following command:

```
mkdir demoapp && cd demoapp
```

Next, initialize npm in the project directory by running:

```
npm init -y
```

Next, you need to install some dependencies. In this tutorial you will need:

1. **Express** : Express is a Node.js framework that provides a powerful feature set for web and mobile applications. Express makes it easier to build backend applications with Node.js.
2. **Joi** : Joi is a data validation library for Node.js.

Install the dependencies with the node package manager by running the following command:

```
npm install express joi
```

Next, create an **index.js** file in the root directory and add the following code block to it:

```
const express = require("express"); const router = require("./routes"); const po
```

The above code block sets up a simple Express server. It configures middleware to analyze incoming query data, process incoming queries, and start the server to listen for incoming queries on port 3000.

Routing and handling requests

The simplest method is that you would create a middleware that processes the query, returning a status code, along with the query body, as a response to any request that tries to send data to your application.

Create the **handler.js** file in the project root directory and add the code block below:

```
const demoHandler = (req, res, next) => { res.send({ code: 201, data: req.body,
```

Next, create the file **router.js** in your project's root directory and add the following code block to your file:

```
const express = require("express"); const demoHandler = require("./handler"); co
```

Create Joi Schema

A Joi schema represents an expected structure of a particular data object and validation rules.

To create a Joi schema, you can use the Joi.object() method and chain the various validation rules given by Joi to define the structure & validation requirements for your data.

For example:

```
const exampleSchema = Joi.object({ name: Joi.string().min(3).required(), });
```

The above example describes a simple Joi schema with a **name** attribute . The name attribute has the value of **Joi.string().min(3).required()** . This means that the name value will be a string, with the required minimum length of 3 characters.

Using Joi, you can chain different methods to add more validation limits for each field defined in the schema.

For example:

```
const userSchema = Joi.object({ email: Joi.string().email().required(), password
```

userSchema defines the following binding requirements for each property:

1. **Email** : Must be a valid email string.
2. **Password** : Must be a string with a 6-character number.
3. **Age** : Several options with a minimum value of 18.
4. **Employed** : An optional boolean.
5. **Phone** : A requested string that matches a specific regular expression **(/^d{3}-d{3}-d{4}$/)** .
6. **Address** : An object representing the user's address with the following extra properties:
   1. **Street** : A required string of at least 3 characters in length.
   2. **City** : A required string of at least 3 characters in length.
   3. **State** : A required string of at least 3 characters in length.
   4. **zip** : A required string with a minimum value of 3.
7. **Hobbies** : Required array of strings.

In addition to the limitations, **userSchema sets the abortEarly** option to **false** . By default, Joi stops running the program as soon as it encounters the first error and prints the error to the console. However, setting this option to **false** ensures Joi checks the entire schema and prints all errors encountered to the console.

Data validation with Joi

Create **validation.js** file and add **userSchema** code to it.

For example:

```
//validation.js const Joi = require("joi"); const userSchema = Joi.object({ //.
```

Then create middleware that intercepts the query payload and verifies them against the provided schema by adding the following code below the **userSchema** code .

```
const validationMiddleware = (schema) => { return (req, res, next) => { const { e
? lý l?i xác th?
c console.log(error.message); res.status(400).json({ errors: error.details }); }
? li?u h?p l?, ti?p t?c x? lý middleware ti?p theo next(); } }; };
```

When a request is made, the middleware will call the schema's **validate** method to validate the query content. If any authentication failure occurs, **the middleware sends a 400 Bad Request** response with the error message retrieved from the authentication failure details.

Otherwise, if the validation succeeds without error, the middeware calls the **next()** function .

Finally, export **the validationMiddleware** and **userSchema** .

```
module.exports = { userSchema, validationMiddleware, };
```

Check the validation conditions

Import **validationMiddleware** and **userSchema** into the **router.js** file , and set up the middleware as follows:

```
const { validationMiddleware, userSchema } = require("./validation"); router.post
```

Start the application by running the command below:

```
node index.js
```

Then create an HTTP POST query to **localhost:3000/signup** using the test data below. You can achieve this using cURL or any other client API.

```
{ "email": "user@example", // Invalid email format "password": "pass", // Passwo
```

This query will fail and return an error object when the payload contains a lot of invalid fields, such as email, password, age, and phone number. Using the provided error object, you can handle the appropriate error.

Simplify data validation with Joi

Here's what you need to know about data validation using Joi. As you can see, it's pretty simple, isn't it? Good luck!

You finished reading the article "**Schema validation in Node.js using Joi**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.

© 2019 TipsMake.com