Using JSON Schema for JSON Validation

Mahesh Samarasinghe • 9 min read • Dec 14, 2023 • Updated

JSON, which stands for **JavaScript Object Notation**, is a lightweight data interchange format. It provides a text-based format readable by humans and machines, making it a prevalent choice among developers for data interchange. However, due to the flexibility of JSON documents, it is easy to misinterpret JSON documents, which can result in miserable application failures. JSON schema helps us to avoid such system failures.

Introduction to JSON schema

JSON schema is a declarative language that allows users to annotate and validate JSON documents. JSON schema has three main goals:

- **Validation**: Validates a JSON document's structure and data types based on a given criterion. This criterion is asserted using the keywords in the JSON schema specification.
- **Documentation**: Serves as documentation for JSON documents used in an application.
- **Hyperlinking**: Connects parts of the JSON data with JSON schema by creating hyperlinks.

In this article, we will primarily focus on implementing JSON data validation using JSON schema.

Why JSON schema

The following JSON object is output from the **Google Distance Matrix API**.

Copy

```
{
   "destination_addresses": [
    "Philadelphia, PA, USA"
   ],
   "origin_addresses": [
    "New York, NY, USA"
   ],
   "rows": [{
    "elements": [{
     "distance": {
      "text": "94.6 mi",
      "value": 152193
     },
      "duration": {
       "text": "1 hour 44 mins",
       "value": 6227
      },
      "status": "OK"
    }]
   }],
   "status": "OK"
 }
```

As you can see, the above JSON document consists of nested objects and arrays. There can be more complicated scenarios. Additionally, applications often need to validate the JSON objects they receive. However, without a properly defined schema, an application cannot validate a JSON document's content.

Take a look at the following code example for JSON schema.

Copy

```
 {
   "$schema":"https://json-schema.org/draft/2020-12/schema",
   "$id":"https://example.com/person.schema.json",
   "title":"Person",
   "description":"A person",
   "type":"object",
   "properties":{
     "name":{
      "description":"Person name",
      "type":"string"
      },
      “age”:{
       “description”:”Person age”,
       “type”:”number”,
       “minimum”:0,
       “maximum”:100
      }
    },
    “required”:[“name”,”age”]
  }
```

The following is a simple JSON object that satisfies the above schema.

Copy

```
{
 "name": "John Doe",
 "age": 27
}
```

JSON schema helps an application understand its JSON objects, their attributes, and the type of the attributes in a better manner. As a result, the application can understand and use the given data without any unexpected failures.

Getting started with JSON schema

To understand how to define a JSON schema, let’s create a sample schema for the following JSON object.

```
{
 "name": "John Doe",
 "email": "john@doe.me",
 "age": 27
}
```

We use five properties known as **JSON keys** to start a schema definition:

- **$schema**: States the draft on which the JSON document is based.
- **$id**: Defines the base URI for the schema. Other URI references in the schema are resolved against this.

- **title** and **description**: Only descriptive values and do not add any restrictions to the data being validated.
- **type**: States the kind of data the schema is defining. This is also the first constraint on the JSON document.

The following is a JSON schema created for a JSON-based employee catalog.

```
{
  "$schema":"https://json-schema.org/draft/2020-12/schema",
  "$id":"https://example.com/employee.schema.json",
  "title":"Employee",
  "description":"An employee in the company",
  "type":"object"
}
```

Next, you need to define the attributes of the object. To do so, add the **properties** validation keyword to the schema. For example, in the following schema definition, we'll add a key called **name** along with a **description** and **type** as **string**.

```
{
  "$schema":"https://json-schema.org/draft/2020-12/schema",
  "$id":"https://example.com/employee.schema.json",
    "title":"Employee",
    "description":"An employee in the company",
    "type":"object",
    "properties": {
      "name": {
        "description": "Name of the employee",
        "type": "string"
      }
    }
  }
```

Furthermore, we should define the employee’s name as a required element, as there cannot be an employee without a name. We can define this validation in the JSON schema using the **required** validation keyword. As shown in the code below, the **name** key can be set as required by adding it inside **required**.

Copy

```
  {
    "$schema":"https://json-schema.org/draft/2020-12/schema",
    "$id":"https://example.com/employee.schema.json",
    "title":"Employee",
    "description":"An employee in the company",
    "type":"object",
    "properties": {
      "name": {
        "description": "Name of the employee",
        "type": "string"
      }
    },
    "required": [ "name" ]
  }
```

Similarly, using the JSON schema, you can define more than one property as a required property. In the following code example, we marked the employee’s email and age as required in addition to the name.

Copy

```
{
  "$schema":"https://json-schema.org/draft/2020-12/schema",
  "$id":"https://example.com/employee.schema.json",
  "title":"Employee",
  "description":"An employee in the company",
  "type":"object",
  "properties": {
    "name": {
      "description": "Name of the employee",
      "type": "string"
     },
    "email": {
      "description": "Email address of the employee",
      "type": "string"
     },
    "age": {
      "description": "Age of the employee",
      "type": "integer"
     }
   },
  "required": [ "name", "email", "age" ]
}
```

JSON schema provides the ability to define many other validations. For example, let us consider the minimum age required to be employed as 18 while the maximum is 60. Here, you can use the **minimum** and **maximum** keywords to enforce the age value to be between 18 and 60.

Refer to the following code example.

Copy

```
{
  "$schema":"https://json-schema.org/draft/2020-12/schema",
  "$id":"https://example.com/employee.schema.json",
  "title":"Employee",
  "description":"An employee in the company",
  "type":"object",
  "properties": {
    "name": {
      "description": "Name of the employee",
      "type": "string"
     },
    "email": {
      "description": "Email address of the employee",
      "type": "string"
     },
    "age": {
      "description": "Age of the employee",
        "type": "integer",
        "minimum": 18,
        "maximum": 60
      }
    },
    "required": [ "name", "email", "age" ]
  }
```

JSON documents do not always contain flat structures. It can also contain arrays or nested data structures. For example, let us update our employee object with a contact number key where an employee can have multiple contact numbers and an address key that can consist of nested values (postal code, street, city).

Copy

```
{
 "name": "John Doe",
 "email": "john@doe.me",
 "age": 27,
 "contactNo": ["+1234567890", "+0987654321"],
 "address": {
      "postalCode": 1111,
      "street": "This street",
      "city": "This city"
    }
  }
```

Furthermore, JSON schema allows the developer to add various validations, such as restricting the element’s data type, the minimum number of elements in the array, whether the array can contain unique items, and so on when using arrays. In our example, assume the **contactNo** key should have at least one value, and there cannot be any duplicate values in the array. You can use the **minItems** and **uniqueItems** keywords provided by the JSON schema to add these validations.

Copy

```
  {
    "$schema":"https://json-schema.org/draft/2020-12/schema",
    "$id":"https://example.com/employee.schema.json",
    "title":"Employee",
    "description":"An employee in the company",
    "type":"object",
    "properties": {
      "name": {
        "description": "Name of the employee",
        "type": "string"
       },
      "email": {
        "description": "Email address of the employee",
        "type": "string"
      },
      "age": {
        "description": "Age of the employee",
        "type": "integer",
        "minimum": 18,
        "maximum": 60
      },
      "contactNo": {
        "description": "Contact numbers of the employee",
        "type": "array",
        "items": {
          "type": "string"
        },
        "minItems": 1,
        "uniqueItems": true
      }
    },
  "required": [ "name", "email", "age" ]
}
```

We can define the Nested objects in a JSON schema using the above-mentioned concepts. Since the value of the **type** validation for the nested structure is **object**, you can use the **properties** keyword to specify the nested object's structure as follows.

Copy

```
{
  "$schema":"https://json-schema.org/draft/2020-12/schema",
    "$id":"https://example.com/employee.schema.json",
    "title":"Employee",
    "description":"An employee in the company",
    "type":"object",
    "properties": {
      "name": {
        "description": "Name of the employee",
        "type": "string"
       },
      "email": {
        "description": "Email address of the employee",
        "type": "string"
       },
      "age": {
        "description": "Age of the employee",
        "type": "integer",
        "minimum": 18,
        "maximum": 60
       },
      "contactNo": {
        "description": "Contact numbers of the employee",
        "type": "array",
        "items": {
          "type": "string"
         },
        "minItems": 1,
        "uniqueItems": true
       },
      "address": {
       "description": "Address of the employee",
       "type": "object",
       "properties": {
         "postalCode": {
          "type": "number"
         },
         "street": {
           "type": "string"
         },
         "city": {
           "type": "string"
         }
       },
       "required": [ "postalCode", "street", "city" ]
     }
   },
   "required": [ "name", "email", "age", "address" ]
 }
```

In the above code, the scope of the **required** validation applies only to the **address** key but not beyond that. Therefore, to add **required** validation to a nested structure, we have to add it inside the nested structure.

JS

Syncfusion JavaScript controls allow you to build powerful line-of-business applications.

Try Now

Advantages of JSON schema

- A properly defined JSON schema makes the JSON document intelligible for humans and computers.
- It provides documentation for JSON documents.
- It provides an easy way of validating JSON objects in an application, enabling interoperability across programming languages by maintaining consistency.
- Prewritten libraries are available for almost all popular programming languages to implement JSON schema in your application. You can find further details about a library for your preferred language here.

Conclusion

In this article, I’ve discussed JSON schemas and how to perform JSON validation using them. JSON schema helps you use the JSON data format confidently, allowing you to validate your JSON structure and ensure it meets API requirements.

I hope you found this article helpful. Thank you for reading it!

Syncfusion's Essential JS 2 is the only suite you will need to build an app. It contains over 80 high-performance, lightweight, modular, and responsive UI components in a single package. Download a free trial to evaluate the controls today.

If you have any questions or comments, you can contact us through our support forums, support portal, or feedback portal. We are always happy to assist you!

Related blogs

- 10 JavaScript Naming Conventions Every Developer Should Know
- 10 Tips and Tricks to Handle JavaScript Objects
- JavaScript Debounce vs. Throttle
- Null vs. Undefined in JavaScript

MEET THE AUTHOR

Mahesh Samarasinghe

I am a full-stack software engineer with over two years of experience working in the MERN stack and AWS. I am also an AWS Community Builder and a content writer in multiple platforms.