T2: JSON

lunes, 12 de mayo de 2025 23:06

Notas:

JSON con JS:

Para convertir un objeto JSON a texto
$ let text = JSON.stringify(obj);

Para convertir un texto a JSON
$ let obj= JSON.parse(text);

Para recorrer un objeto JSON
$ for(let key in jsonObj){
console.log("key: "+ key + ", value: " + jsonObj[key]);
}

JSON con JS – Ejemplo

```
let persona = {"nombre": "Juan", "edad": 23};
console.log("Persona:\n", persona);
let text = JSON.stringify(persona);
console.log("\nText:\n", text);
let obj = JSON.parse(text);
console.log("\nObj:\n", obj);
console.log("\nKeys:")
for(let key in obj) {
    console.log("key:"+ key +", value:"+ obj[key]);
}
console.log("\nStringify:\n", JSON.stringify(persona, null, 2));
```

JSON Schema

Es el vocabulario para anotar y validar documentos JSON.
Con una estructura básica:

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/product.schema.json",
    "title": "Product",
    "description": "A product in the catalog",
    "type": "object",
    "properties": {
        "productId": {
            "type": "number",
            "description": "The unique identifier for a product"
        },
    "required": ["productId"]
}
```

Cuando un esquema se empieza a volver muy complejo, lo ideal es definir **Sub-Schemas** con las etiquetas *$def* y *$ref*

```
{
...
    "properties": {
        "first name": {"$ref": "#/$defs/name"},
        "last name" :{"$ref": "#$defs/name"}
    }
},
"required" : ["first name", "last name"],
"$defs": {
    "name": {"type:" "string"}
}
```

El *JSON Type Definition JTD*, es un tipo de JSON Schema más ligero y es el estándar oficial más reciente (2020) que permite describir la forma de los datos.

Ofrece **ventajas** como:
- Compatibilidad con muchos lenguajes, y validadores
- Simple y con buenas prácticas (sin ambigüedades -como las que suele causar *anyOf* o *not* en el JSON Schema-)
- Define la FORMA del dato, no solo sus restricciones
- Soporta bien los *tagged unions*
- Permite compilar parsers y serializers
- Es robusto a errores comunes

Sin embargo por consecuencia sus **desventajas** directas son:
- Es limitado en comparación
  - No soporta *Untagged unions* (que es cuando podemos definir que un valor puede tener varios tipos)
  - No tiene condicionales
  - No permite hacer referencias entre esquemas
- No tiene meta-schema (que es un validador general de los schemas)

También se pueden definir rangos (numéricos, strings, arryas)

```
} ...
"type" : "number",
"minimun": 0,
"maximum": 2
... }
} ...
"type" : "number",
"exclusiveMinimun": 0,
"exclusiveMaximum": 2
... }
} ...
"type" : "number",
"multipleOf": 5
... }
```

----> Acepta [0:2]

----> Acepta el [1]

----> Acepta múltiplos de [5]

- minLength
- maxLength
- format: unos formatos predefinidos. Ej:
  - date-time
  - email
  - uri
- pattern: permite definir una expresión regular que valide el string

array:
- minItems
- maxItems
- uniqueItems: no se pueden repetir elementos si vale true
- items: para definir las características de todos los ítems
- prefixItems: cuando importa el orden
- contains: para que al menos un elemento sea del tipo especificado
- minContains / maxContains: usado junto con contains

Además de los diferentes tipos, hay también diferentes formas de definirlas:

Por ejemplo, si un elemento puede admitir varios **tipos**, lo definiremos como un ***array*** de las posibilidades:

```
{
  "type": ["number", "string"]
}
```

42 ✅
"42" ✅
true ❌

Otro caso es definirlo como ***enum*** para cuando hay solo un "catalogo" restringido de **valores**:

```
{
  "enum": ["red", "amber", "green", null, 42]
}
```

"red" ✅
42 ✅
null ✅
"blue" ❌

Finalmente se pueden definir también como ***const*** para valores **constantes:**

```
{
  "const": "Spain"
}
```

"Spain" ✅
"spain" ❌

Los *types* principales para los esquemas son: [string, number, object, array, boolean, null]

Sería equivalente al JSON:

```
{
    "productId": 1
}
```

- Hay muchas más etiquetas:
  - deprecated
  - readOnly
  - writeOnly
  - $comment
  - default
  - examples: array de valores válidos
  - additionalProperties:
    - true/false (valor por defecto true)
    - Restricciones adicionales que tienen que cumplir las propiedades extras

Esta "herramienta" proporciona ***ventajas***:
- Como la estandarización y normalización ya que es ampliamente adoptado
  - Compatible con OpenAPI
- Soporta validaciones -y restricciones- complejas (como que en un dato deba cumplir varias condiciones; validaciones condicionales; dependencias entre propiedades; semánticas y de formatos)
- Es óptimo para validar objetos JS o archivos de configuración (como lo es el package.json por ejemplo)

json Copiar código

```
"if": { "properties": { "tipo": { "const": "profesor" } } },
"then": { "required": ["departamento"] }
```

json

```
"dependencies": {
  "tarjetaCredito": ["nombreTitular"]
}
```

```
{ "type": "string", "format": "email" }
```

→ Acepta solo si el string es un email válido.

Otros formatos útiles:
- `date`, `date-time`
- `ipv4`, `hostname`
- `uri`, `uuid`

Puedes validar que un dato cumpla **una de varias condiciones** (`anyOf`, `oneOf`, `not`, `allOf`).

json Copiar código

```
"anyOf": [
  { "type": "string" },
  { "type": "number" }
]
```

```
"anyOf": [
  { "type": "number", "minimum": 0 },
  { "type": "string", "maxLength": 10 }
]
```

... Y aunque todo suena muy bien, también existen ***desventajas***/inconvenientes:
- Se dice/entiende que el Schema NO define la FORMA del dato, sino, las restricciones que este debe cumplir... Esto lo hace menos portable y legible
- *No standard support for tagged unions:* Se refiere a que cuando se tienen varios posibles "tipos de objeto" y se usa un campo "type" para diferenciarlo, es complicado y engorroso (incorporar validaciones condicionales que lo vuelven poco legible)
- Es complejo... al punto de que se suele prescindir de las opciones avanzadas que este

Para validar JSON y sus esquemas: https://www.jsonschemavalidator.net/

propone

- El estándar de JSON Schema aún no ha sido publicado como un RFC oficial, por lo que es propenso a cambiar

Integración de NODE.JS y JSON

Aunque existen varias bibliotecas -como *jsonschema*, *djv*- para validar objetos JSON contra un JSON schema, ***AJV*** es el *another* JSON validador, que soporta su última versión.

```
$ npm install ajv
$ npm install ajv-formats
```

Ajv – Ejercicio

- Crea un servicio web que valide al menos dos JSON diferentes que reciba por dos rutas POST. Para cada ruta:
  - Que devuelva un 200 si el JSON es válido
  - Que devuelva un 400 si el JSON no es válido o por cualquier otro error
- Guarda cada schema en su archivo `.json`
- Usa Postman para comprobar que la aplicación funciona
- Añadir en la actividad un el enlace al repositorio

app.js M app.js\...

```js
const express = require('express');
const fs = require('fs');

const app = express();
app.use(express.json());

const Ajv = require("ajv");
const addFormats = require("ajv-formats");
const ajv = new Ajv({strict: false, validateSchema: false});

addFormats(ajv)

const raceSchema = require("./schemas/race_schema.json");
const rutineSchema = require("./schemas/gym_rutine_schema.json");

app.post('/race', (req, res) => {
  const validateRace = ajv.compile(raceSchema);
  const isValid = validateRace(req.body);

  if (isValid) {
    res.status(200).send("Race JSON is valid");
  } else {
    console.log(validateRace.errors);
    res.status(400).send("Race JSON is invalid");
  }
});

app.post('/rutine', (req, res) => {
  const validateRutine = ajv.compile(rutineSchema);
  const isValid = validateRutine(req.body);

  if (isValid) {
    res.status(200).send("Rutine JSON is valid");
  } else {
    console.log(validateRutine.errors);
    res.status(400).send("Rutine JSON is invalid");
  }
});


app.listen(3000, () => {
  console.log('Servidor escuchando en puerto 3000');
});

```