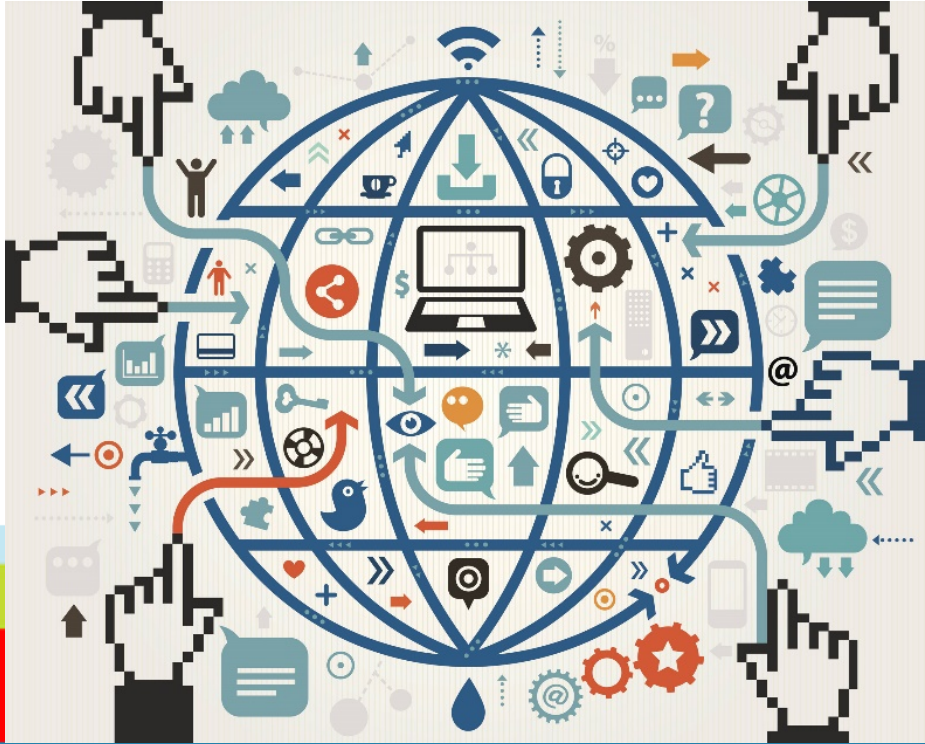
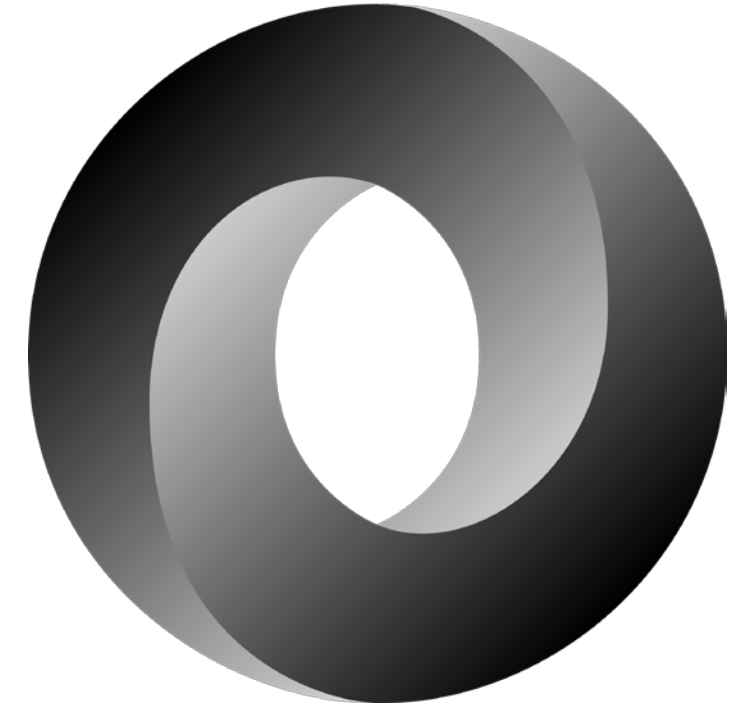


CEU

*Universidad
San Pablo*

JSON



Sistemas Web II

Grado en Ingeniería de Sistemas de Información

Álvaro Sánchez Picot

alvaro.sanchezpicot@ceu.es

v20230309

JSON

- JavaScript Object Notation
- Formato ligero para el intercambio de datos
- Basado en el estándar ECMA-262 de 1999
 - Versión más reciente [ECMA-404](#) de 2017
- Soporta objetos, arrays y otros valores
- No permite comentarios
- [Más información](#)
- [Validador](#)
- [Guía de estilo](#)

JSON – Ejemplo

```
{  
  "firstName": "John",  
  "lastName": "Smith",  
  "isAlive": true,  
  "age": 27,  
  "address": {  
    "streetAddress": "21 2nd Street",  
    "city": "New York",  
    "state": "NY",  
    "postalCode": "10021-3100"  
  },  
  "phoneNumbers": [  

```

```
  
    {  
      "type": "home",  
      "number": "212 555-1234"  
    },  
    {  
      "type": "office",  
      "number": "646 555-4567"  
    }  
  ],  
  "children": [],  
  "spouse": null  
}
```

JSON – Sintaxis

Objeto:

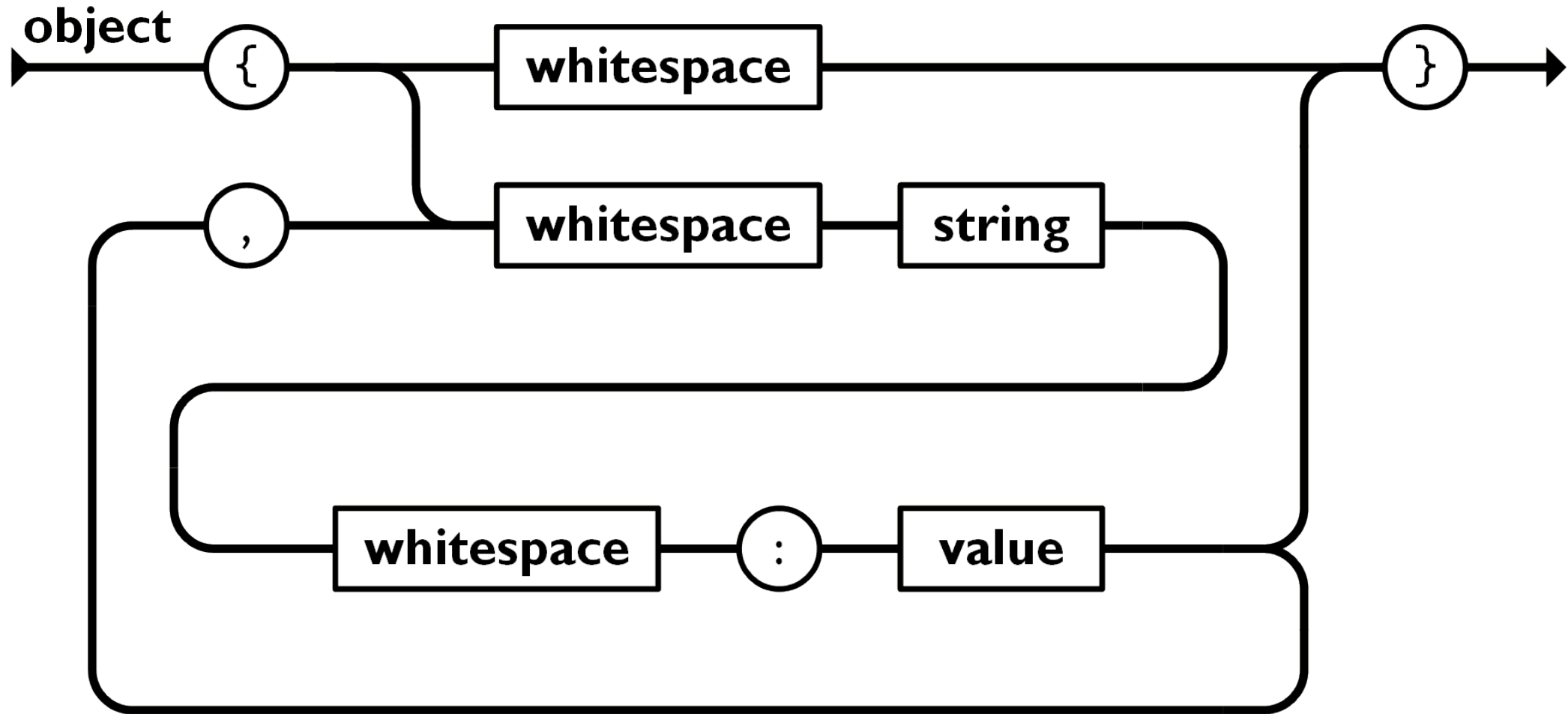
- Set desordenado de pares clave / valor:
 `"clave": valor`
- Rodeado por llaves: `{ y }`
- La clave es un String y tiene que tener comillas dobles
- La clave debería usar nomenclatura lower camel case
- Los datos se separan con comas
 - Cuidado con las trailing commas
- Puede estar vacío: `{ }`

JSON – Sintaxis

Objeto:

```
{  
  "nombreCompleto": "Juan Pérez Rodríguez",  
  "edad": 27      ← Cuidado con la trailing comma  
}
```

JSON – Sintaxis



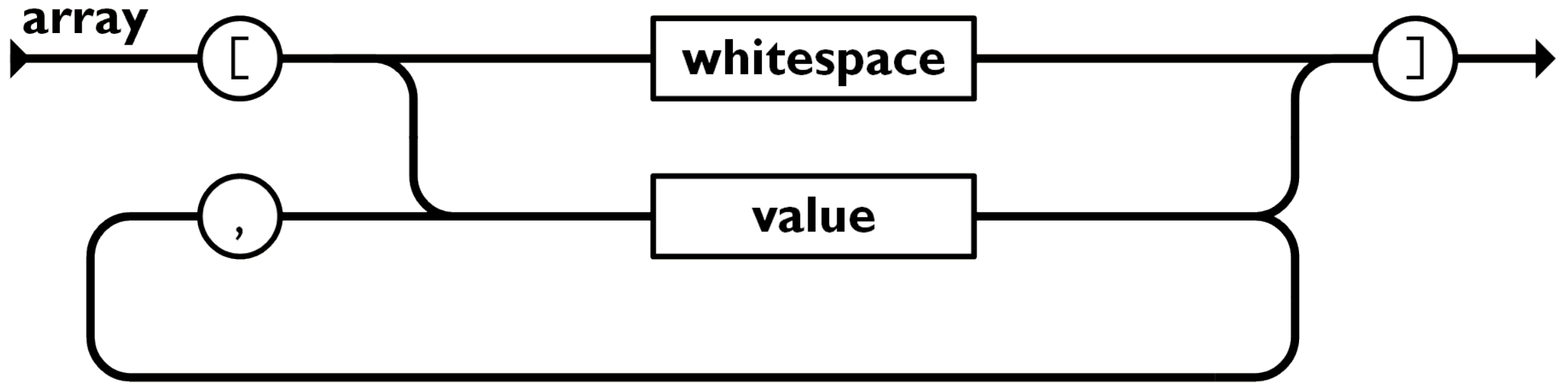
JSON – Sintaxis

Array:

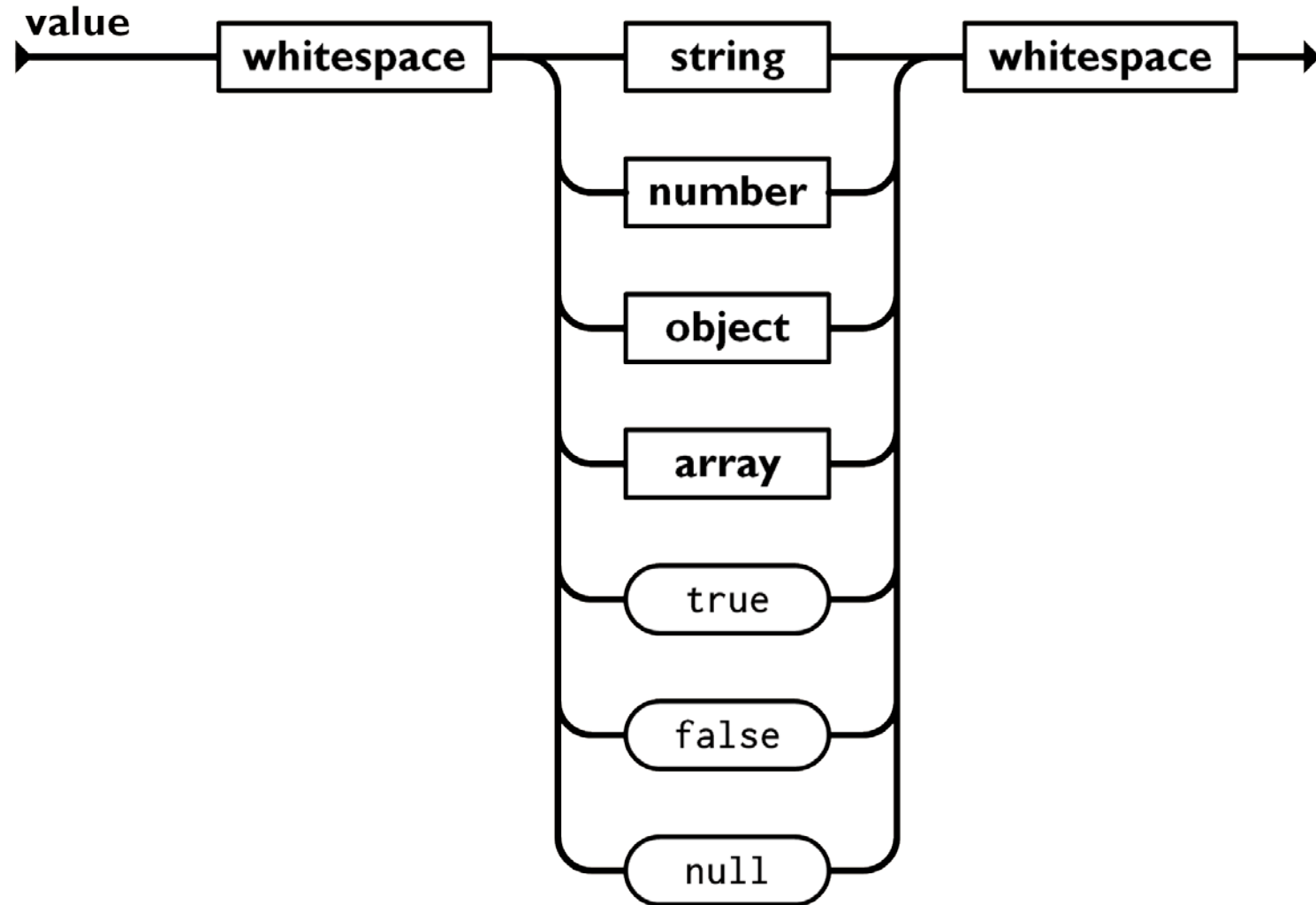
- Colección ordenada de valores
- Rodeado por corchetes: [y]

```
[  
  {"movil": 612345678},  
  {"fijo": 912345678}  
]
```

JSON – Sintaxis



JSON – Sintaxis



JSON – Sintaxis

- Se pueden anidar elementos:

```
{  
  "nombre": "Juan",  
  "direccion": {  
    "calle": "Avenida Ciudad de Barcelona 23",  
    "ciudad": "Madrid"  
  },  
  "telefonos": [  
    {"movil": 612345678},  
    {"fijo": 912345678}  
  ],  
  "edad": 27  
}
```

JSON – Sintaxis

- Tiene que haber un único elemento raíz que tiene que ser un valor

```
[  
  {"nombre" : "Juan"},  
  {"nombre" : "Ana"},  
  {"nombre" : "Sofía"},  
  {"nombre" : "Andrés"}  
]
```

JSON



- No se garantiza el orden del contenido de los objetos

```
{  
  "nombre": "Juan",  
  "edad": 27  
}  
  
↔  
  
{  
  "edad": 27,  
  "nombre": "Juan"  
}
```

JSON



- Los elementos de un array sí que están ordenados

```
[  
  {"nombre" : "Juan"},  
  {"nombre" : "Ana"}  
]  
  
↔  
  
{  
  "edad": 27,  
  "nombre": "Juan"  
}
```

JSON – Ejercicio

- ¿Está bien formado el siguiente JSON?

```
{  
  "nombre": "Juan",  
  "telefonos": [  
    {"movil": 612345678},  
    {"fijo": [912345678]}, false  
  ]  
  edad: 27, mayorDeEdad: "true",  
  "direccion": {  
    "calle": "Avenida Ciudad de Barcelona 23",  
    "ciudad": Madrid, null  
  },  
}
```

JSON con JS

- Paquete JSON
 - No es específico de Node.js
- Convertir un objeto JSON a texto

```
let text = JSON.stringify(obj);
```
- Convertir un texto en formato JSON a texto

```
let obj = JSON.parse(text);
```

JSON con JS

- 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

JSON Schema

- Vocabulario para anotar y validar documentos JSON
- Versión más reciente: [Draft 2020-12](#)
- En proceso de estandarización por el IETF
- [Más información](#)

JSON Schema

- 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": {...}  
  "required": [...]  
}
```

JSON Schema

- Estructura básica:
 - `$schema`: Versión de JSON Schema
 - `$id`: URI base del esquema
 - `title` y `description`: anotaciones descriptivas sin implicaciones en la validación
 - `type`: valor del elemento raíz
 - `properties`: objeto con las keywords
 - `required`: array de propiedades que son obligatorias

JSON Schema

- `properties`:
 - Objeto con las keywords de nuestro objeto
 - Para cada keyword:
 - `description`: anotación con la descripción
 - `type`: valor del elemento

JSON Schema – Ejemplo

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

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

JSON Schema

- type:
 - "string"
 - "number": tanto números enteros como con decimales
 - "integer": pueden ser decimales que terminen en 0: 1.0
 - "object"
 - "array"
 - "boolean": true o false, sin comillas
 - "null": null
- [Más información](#)

JSON Schema

- type también puede ser un array si acepta varios tipos:

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

- En vez de type podemos usar enum para listar los valores válidos

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

- Si solo admitimos un valor podemos usar const:

`"const": "Spain"`

JSON Schema – Ejemplo

```
{ (...)
  "properties": {
    (...)
    "productName": {
      "description": "Name of the product",
      "type": "string"
    },
    "price": {
      "description": "The price of the product",
      "type": "number",
      "exclusiveMinimum": 0
    }
  },
  "required": [ "productId", "productName", "price" ]
}
```

```
{
  "productId": 1,
  "productName": "A green door",
  "price": 12.50
}
```

JSON Schema

- Rangos de number e integer:
 - minimum: incluyendo el valor
 - exclusiveMinimum (*)
 - maximum: incluyendo el valor
 - exclusiveMaximum (*)
- Múltiplos:
 - multipleOf: Restringirlo a un múltiplo de un valor
- [Más información](#)
- (*) En versiones previas eran un boolean

JSON Schema

- Para strings:
 - minLength
 - maxLength
 - format: unos formatos predefinidos. Ej:
 - date-time
 - email
 - uri
 - pattern: permite definir una expresión regular que valide el string
 - [Más información](#)

JSON Schema – Ejemplo

```
{ (...)  
  "properties": {  
    (...)  
    "tags": {  
      "description": "Tags for the product",  
      "type": "array",  
      "items": {  
        "type": "string"  
      },  
      "minItems": 1,  
      "uniqueItems": true  
    },  
  },  
  "required": [ "productId", "productName", "price" ]  
}
```

```
{  
  "productId": 1,  
  "productName": "A green door",  
  "price": 12.50,  
  "tags": [ "home", "green" ]  
}
```

JSON Schema

- 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
- [Más información](#)

JSON Schema – Ejemplo

```
{ (...)  
  "properties": {  
    (...)  
    "dimensions": {  
      "type": "object",  
      "properties": {  
        "length": { "type": "number" },  
        "width": { "type": "number" },  
        "height": { "type": "number" }  
      },  
      "required": [ "length", "width", "height" ]  
    },  
    "required": [ "productId", "productName", "price" ]  
  }  
}
```

```
{  
  "productId": 2,  
  "productName": "Statue",  
  "price": 17.59,  
  "tags": [ "cold", "ice" ],  
  "dimensions": {  
    "length": 7.0,  
    "width": 12.0,  
    "height": 9.5  
  }  
}
```

JSON Schema – Ejemplo

```
{ (...)  
  "properties": {  
    (...)  
    "warehouseLocation": {  
      "description": "Coordinates of the warehouse where  
the product is located.",  
      "$ref": "https://example.com/geographical-  
location.schema.json"  
    }  
  },  
  "required": [ "productId", "productName", "price" ]  
}
```

```
{  
  "productId": 2,  
  "productName": "Statue",  
  "price": 17.59,  
  "tags": [ "cold", "ice" ],  
  "dimensions": {  
    "length": 7.0,  
    "width": 12.0,  
    "height": 9.5  
  },  
  "warehouseLocation": {  
    "latitude": -78.75,  
    "longitude": 20.4  
  }  
}
```


JSON Schema

```
{
  "$id": "https://example.com/geographical-location.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Longitude and Latitude",
  "description": "A geographical coordinate on a planet (most commonly Earth).",
  "required": [ "latitude", "longitude" ],
  "type": "object",
  "properties": {
    "latitude": {
      "type": "number",
      "minimum": -90,
      "maximum": 90
    },
    "longitude": {
      "type": "number",
      "minimum": -180,
      "maximum": 180
    }
  }
}
```



JSON Schema

- Podemos definir subesquemas con la etiqueta `$defs` y referenciarlos con `$ref`

```
{ (...)  
  "properties": {  
    "first_name": { "$ref": "#/$defs/name" },  
    "last_name": { "$ref": "#/$defs/name" }  
  },  
  "required": ["first_name", "last_name"],  
  "$defs": {  
    "name": { "type": "string" }  
  }  
}
```

JSON Schema

- 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

JSON Schema – Referencias

- Especificación
- Getting started
- Implementaciones
 - Validador

JSON Schema – Ejercicio



Crea un JSON que sea válido con el siguiente JSON Schema:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema#",
  "$id": "http://example.com/schemas/painting.schema.json",
  "type": "object",
  "title": "Painting",
  "description": "Painting information",
  "required": ["name", "artist", "dimension", "description", "tags"],
  "properties": {
    "name": {
      "type": "string",
      "description": "Painting name"
    },
    "artist": {
      "type": "string",
      "maxLength": 50,
      "description": "Name of the artist"
    },
    "description": {
      "type": ["string", "null"],
      "description": "Painting description"
    },
    "dimension": {
      "$ref": "#/defs/dimension"
    },
    "tags": {
      "type": "array",
      "items": { "$ref": "#/defs/tag" }
    }
  },
  "$defs": {
    "tag": {
      "type": "string",
      "enum": ["oil", "watercolor", "digital", "famous"]
    },
    "dimension": {
      "type": "object",
      "title": "Painting dimension",
      "description": "Describes the dimension of a painting in cm",
      "required": ["width", "height"],
      "properties": {
        "width": { "type": "number", "description": "Width of the product", "minimum": 1 },
        "height": { "type": "number", "description": "Height of the product", "minimum": 1 }
      }
    }
  }
}
```

```
{
  "name": "The Starry Night",
  "artist": "J.M.W. Turner",
  "description": "A reproduction of a famous painting",
  "dimension": {
    "width": 100,
    "height": 100
  },
  "tags": [
    "oil",
    "famous"
  ]
}
```

[Source](#)

JSON Schema – Ejercicio



Crea un JSON Schema que sea válido para el siguiente JSON:

```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": [
        "Radiation resistance",
```

```
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name": "Madame Uppercut",
      "age": 39,
      "secretIdentity": "Jane Wilson",
      "powers": [
        "Million tonne punch",
        "Damage resistance"
      ]
    }
  ]
}
```

[Source](#)

JSON Schema – Ejercicio



Realiza los siguientes pasos:

- Piensa en la información que podríamos guardar sobre una película específica
- Crea un objeto JSON con la información anterior
- Diseña el JSON Schema para validar el JSON anterior

Puedes usar páginas como [Imdb](#) o [Rotten Tomatoes](#).



JSON Schema – Ventajas

- Wide specification adoption
- Used as part of OpenAPI specification
- Support of complex validation scenarios:
 - untagged unions and boolean logic
 - conditional schemas and dependencies
 - restrictions on the number ranges and the size of strings, arrays and objects
 - semantic validation with formats, patterns and content keywords
 - distribute strict record definitions across multiple schemas (with unevaluatedProperties)
- Can be effectively used for validation of any JavaScript objects and configuration files

JSON Schema – Inconvenientes

- Defines the collection of restrictions on the data, rather than the shape of the data
- No standard support for tagged unions
- Complex and error prone for the new users (Ajv has strict mode enabled by default to compensate for it, but it is not cross-platform)
- Some parts of specification are difficult to implement, creating the risk of implementations divergence:
 - reference resolution model
 - unevaluatedProperties/unevaluatedItems
 - dynamic recursive references
- Internet draft status (rather than RFC)

[Source](#)

JSON TYPE DEFINITION

JSON Type Definition

- JTD
- Describir la forma de los datos
- Schema ligero
- Estándar más reciente que JSON Schema
- Versión de noviembre 2020: [RFC 8927](#)
- [Más información](#)

JSON Type Definition

```
{  
  "properties": {  
    "name": { "type": "string" },  
    "isAdmin": { "type": "boolean" }  
  },  
  "optionalProperties": {  
    "middleName": { "type": "string" }  
  }  
}
```

```
{  
  "name": "William Sherman",  
  "isAdmin": false,  
  "middleName": "Tecumseh"  
}
```

JSON Type Definition – Ventajas

- Aligned with type systems of many languages - can be used to generate type definitions and efficient parsers and serializers to/from these types
- Very simple, enforcing the best practices for cross-platform JSON API modelling
- Simple to implement, ensuring consistency across implementations
- Defines the shape of JSON data via strictly defined schema forms (rather than the collection of restrictions)
- Effective support for tagged unions
- Designed to protect against user mistakes
- Supports compilation of schemas to efficient serializers and parsers (no need to validate as a separate step)
- Approved as RFC8927

JSON Type Definition – Inconvenientes

- Limited, compared with JSON Schema - no support for untagged unions*, conditionals, references between different schema files**, etc.
- No meta-schema in the specification*
- Brand new - limited industry adoption (as of January 2021)

[Source](#)

NODE.JS

Node.js

- Hay varias librerías para validar JSON:
 - ajv
 - jsonschema
 - djv

ajv

- JSON validator
- Soporta la última versión de JSON Schema ([2020-12](#))
 - También soporta [JTD](#)
- Contribuyen y patrocinan empresas como Mozilla y Microsoft
- [Más información](#)
- [npm](#)

```
npm install ajv
```

Ajv – demo 1

```
const Ajv = require("ajv")
const ajv = new Ajv() // options can be passed, e.g. {allErrors: true}
const schema = {
  type: "object",
  properties: {
    foo: {type: "integer"},
    bar: {type: "string"}
  },
  required: ["foo"],
  additionalProperties: false
}
const validate = ajv.compile(schema)
const data = { foo: 1, bar: "abc"}
const valid = validate(data)
if (!valid) console.log(validate.errors)
```

Ajv – demo 2

//Archivo schemas/coordinate.schema.json

```
{
  "$id": "https://example.com/geographical-location.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Longitude and Latitude Values",
  "description": "A geographical coordinate.",
  "required": [ "latitude", "longitude" ],
  "type": "object",
  "properties": {
    "latitude": {
      "type": "number",
      "minimum": -90,
      "maximum": 90 },
    "longitude": {
      "type": "number",
      "minimum": -180,
      "maximum": 180
    }
  }
}
```

Ajv – demo 2

//Archivo schemas/person.schema.json

```
{  "$id": "person.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Person",
  "type": "object",
  "properties": {
    "firstName": {
      "type": "string",
      "description": "The person's first name."
    },
    "lastName": {
      "type": "string",
      "description": "The person's last name."
    },
    "age": {
      "description": "Age in years which must be equal to or greater than zero.",
      "type": "integer",
      "minimum": 0
    }
  }
}
```



Ajv – demo 2

```
//Archivo schemas/index.js
```

```
const Ajv2020 = require("ajv/dist/2020");
```

```
const ajv = new Ajv2020();
```

```
const schema_person = require("./person.schema.json")
```

```
const schema_coordinate = require("./coordinate.schema.json")
```

```
ajv.addSchema(schema_person, "person")
```

```
ajv.addSchema(schema_coordinate, "coordinate")
```

```
module.exports = ajv;
```

Ajv – demo 2

```
//Archivo index.js
```

```
const ajv = require("./schemas")
const person_json_ok = {  "firstName": "John",  "lastName": "Doe",  "age": 21}
const person_json_error = {  "firstName": "John",  "lastName": 2,  "age": -10}
validatePerson(person_json_ok);
validatePerson(person_json_error);
function validatePerson(json) {
    const validate = ajv.getSchema("person");
    if (validate(json)) {
        console.log("JSON OK");
    } else {
        console.log("JSON NOT OK");
        console.log(validate.errors);
    }
}
```

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

Referencias

- [JSON](#)
- [Estándar Ecma-404](#)
- [JSON Schema](#)
- [Specification](#)
- [JSON Schema Validator](#)