

# Desarrollo de Front-END

## Tema 8

### Json

César Andrés Sánchez

Universidad Camilo José Cela  
Escuela Politécnica Superior de Tecnología y Ciencia

Curso 2025 - 2026



# Json

# ¿Qué es JSON?

- ▶ Formato ligero de intercambio de datos, independiente de lenguaje y plataforma.
- ▶ Estándar abierto: **RFC 8259** (actual) y **ECMA-404**.  
(Reemplaza la referencia histórica RFC 4627).
- ▶ Nace de JavaScript (*JavaScript Object Notation*), pero hoy es agnóstico.
- ▶ Alternativa a XML: más simple y concisa; hoy en día, ampliamente predominante en APIs web.
- ▶ Tipos primitivos: `string`, `number`, `boolean`, `null`. Compuestos: `array`, `object`.

# Reglas de sintaxis (resumen)

- ▶ Claves de objeto: **siempre** cadenas entre comillas dobles.
- ▶ Cadenas: comillas dobles, secuencias de escape Unicode (p. ej. `\u00F1`).
- ▶ Números: enteros o de coma flotante; no hay NaN/Infinity.
- ▶ Booleanos y nulo: `true`, `false`, `null`.
- ▶ Estructuras: `[]` (arrays), `{ }` (objetos), separados por comas.

## Prohibido en JSON

- ▶ Comentarios (`//` o `/* */`)
- ▶ Comillas simples para cadenas
- ▶ Comas finales (*trailing commas*)
- ▶ Valores especiales: `undefined`, `NaN`, `Infinity`
- ▶ Claves duplicadas (comportamiento no definido)

# Ejemplo canónico

```
1 {  
2   "id": 12345,  
3   "nombre": "Juan Perez",  
4   "activo": true,  
5   "roles": ["admin", "editor"],  
6   "perfil": {  
7     "email": "juan@example.com",  
8     "telefono": null  
9   }  
10 }
```



# Números y cadenas

## Números

```
1 0, 42, -12, 3.1415, 6.022e23
```

## Cadenas (escapes)

```
1 "línea\nueva", "tab\tulador"
```

## Consejo

Normaliza a UTF-8 y *no* dependas de escapes Unicode salvo que sea necesario.

# Arrays y objetos

## Array

```
1 [1, "azul", [1,2,3]]
```

## Objeto

```
1 { "nombre": "Juan", "notas": [5.5, 7.2, 6.1]
  }
```

# Convenciones y estilo

- ▶ Claves en `camelCase` o `snake_case` de forma consistente.
- ▶ Nombres claros: evita abreviaturas crípticas.
- ▶ *Pretty print* en desarrollo; compacta en producción.
- ▶ Versiona tu esquema: `"version": 1` o vía ruta/namespace.
- ▶ Documenta contratos: ejemplos + JSON Schema.

# Errores comunes (y cómo evitarlos)

## Incorrecto

```
1 {  
2   'clave': 'valor', // <-- comillas simples  
      y comentario  
3   "lista": [1,2,3,], // <-- coma final  
4 }
```

## Correcto

```
1 {  
2   "clave": "valor",  
3   "lista": [1, 2, 3]  
4 }
```

## Claves duplicadas

Evítalas: algunos parsers se quedan con la última, otros lanzan error.

# Validar con JSON Schema (vocabulario draft 2020-12)

```
1 {
2   "$schema": "https://json-schema.org/draft/2020-12/schema",
3   "$id": "https://example.com/usuario.schema.json",
4   "title": "Usuario",
5   "type": "object",
6   "required": ["id", "nombre"],
7   "properties": {
8     "id": {"type": "integer", "minimum": 1 },
9     "nombre": {"type": "string", "minLength": 1 },
10    "email": {"type": "string", "format": "email" },
11    "roles": {
12      "type": "array",
13      "items": {"type": "string" },
14      "uniqueItems": true
15    }
16  },
17  "additionalProperties": false
18 }
```

# Cuándo validar

- ▶ A la entrada de tu API (*gateway*/controlador): rechaza pronto.
- ▶ Antes de persistir en BD: coherencia interna.
- ▶ En clientes: falla rápido y con mensajes claros.
- ▶ En pipelines ETL: al aterrizar datos o antes de publicar.

## Parseo y serialización

```
1 const obj = JSON.parse(textoJSON); // puede lanzar SyntaxError
2 const txt = JSON.stringify(obj, null, 2); // pretty print
```

## Reviver / Replacer

```
1 const d = JSON.parse(x, (k,v) => (k==="fecha" ? new Date(v) : v));
2 const out = JSON.stringify(d, (k,v) => v instanceof Date ? v.toISOString() : v);
```

## Fetch con JSON

```
1 const resp = await fetch('/api/usuarios', {
2   method: 'POST',
3   headers: { 'Content-Type': 'application/json' },
4   body: JSON.stringify({ nombre: 'Ana' })
5 });
```

# Python (módulo json)

```
1 import json
2
3 # cargar
4 data = json.loads(texto_json) # desde cadena
5 with open('datos.json', encoding='utf-8') as f:
6     data = json.load(f) # desde archivo
7
8 # volcar
9 s = json.dumps(data, ensure_ascii=False) # UTF-8 limpio
10 s_pretty = json.dumps(data, ensure_ascii=False, indent=2)
11
12 # validación simple
13 try:
14     json.loads(texto_json)
15 except json.JSONDecodeError as e:
16     print("JSON inválido:", e)
```

# Contratos de API

```
1 {  
2   "status": "ok",  
3   "data": { "id": 7, "nombre": "Ana" },  
4   "error": null,  
5   "meta": { "requestId": "9f1...", "version": 1 }  
6 }
```

- ▶ Estructuras consistentes facilitan el manejo de errores y el versionado.
- ▶ Incluye meta para trazabilidad y compatibilidad.

# Paginación y filtros

- ▶ Paginación: `page/pageSize` o `limit/offset`.
- ▶ Enlaces HATEOAS opcionales: `self`, `next`, `prev`.
- ▶ Filtros y ordenación declarativos en query params.

# Rendimiento

- ▶ Minimiza espacios en producción: *payloads* más ligeros.
- ▶ Compresión HTTP (gzip/br) y caching con ETag/Cache-Control.
- ▶ Evita *overfetching*: endpoints específicos o GraphQL/JSON:API.
- ▶ Streaming y **NDJSON** (JSON por línea) para grandes volúmenes.

# Seguridad

- ▶ Cabeceras correctas: `Content-Type: application/json`.
- ▶ Protege frente a JSON hijacking (antiguo) y CSRF en peticiones mutantes.
- ▶ Deserializa de forma segura: nunca evalúes texto como código.
- ▶ Limpia entradas: ataques de inyección en consultas subsecuentes.
- ▶ Límites de tamaño y profundidad para evitar *DoS* por parsing.

## JSON vs XML vs YAML (resumen)

	<b>JSON</b>	<b>XML</b>	<b>YAML</b>
Sintaxis	Simple	Verbosa	Muy concisa
Tipos	Limitados	Atributos + texto	Rico/implícito
Comentarios	No	Sí	Sí
Esquemas	JSON Schema	XSD	JSON Schema/YAML
Streaming	Sí (NDJSON)	Sí	Parcial

## Ejercicio 1: Encontrar el error

```
1 {  
2   "id": 1,  
3   "nombre": "Ana",  
4   "tags": ["ventas", "crm",], // <--  
5 }
```

### Solución

Quitar la coma final en el array y en el objeto; eliminar comentarios.

## Ejercicio 2: Normalizar fechas

```
1 // Entrada
2 { "fecha": "21/10/2025 09:30" }
3
4 // Salida deseada (ISO 8601)
5 { "fecha": "2025-10-21T09:30:00+02:00" }
```

- ▶ Define convención ISO 8601 siempre.
- ▶ Usa `reviver/replacer` (JS) o conversión previa (Python).

# Recursos

- ▶ Especificación: ECMA-404 y RFC 8259.
- ▶ JSON Schema: <https://json-schema.org/>
- ▶ Validador online: <https://www.jsonschemavalidator.net/>
- ▶ Lint/Format: <https://jqplay.org/>, <https://jsonlint.com/>

# Resumen

- ▶ JSON es simple, portable y dominante en integraciones.
- ▶ Define contratos y valida con JSON Schema.
- ▶ Cuida estilo, seguridad y rendimiento.
- ▶ Usa NDJSON para *streaming*/logs.

# Procesamiento de JSON en JavaScript

## Funciones nativas

JavaScript proporciona dos funciones fundamentales para trabajar con JSON:

- ▶ `JSON.stringify()` - Convierte objetos a cadenas JSON
- ▶ `JSON.parse()` - Convierte cadenas JSON a objetos

## Compatibilidad

- ▶ Disponibles en todos los navegadores modernos (IE8+)
- ▶ Parte del estándar ECMAScript 5
- ▶ No requieren librerías externas

# JSON.stringify() - Conversión a JSON

## Sintaxis básica

`JSON.stringify(valor[, replacer[, espacio]])`

- ▶ `valor`: Objeto/valor a convertir
- ▶ `replacer` (opcional): Función de filtrado o array
- ▶ `espacio` (opcional): Indentación para formato legible

## Ejemplo básico

```
'use strict';  
let lista = ["sota", "caballo", "rey"];  
console.log(typeof lista, lista);  
// object ["sota", "caballo", "rey"]
```

```
let cadena = JSON.stringify(lista);  
console.log(typeof cadena, cadena);  
// string ["sota","caballo","rey"]
```

# Ejemplos de JSON.stringify()

## Objetos simples

```
const usuario = {  
  nombre: "Ana",  
  edad: 30,  
  activo: true  
};
```

```
console.log(JSON.stringify(usuario));  
// {"nombre":"Ana","edad":30,"activo":true}
```

## Con indentación

```
console.log(JSON.stringify(usuario, null, 2));  
// {  
//   "nombre": "Ana",  
//   "edad": 30,  
//   "activo": true
```

## Valores omitidos

```
const obj = {  
  nombre: "Juan",  
  edad: 25,  
  ciudad: undefined,  
  metodo: function() {}  
};
```

```
console.log(JSON.stringify(obj));  
// {"nombre":"Juan","edad":25}
```

## Arrays

```
const numeros = [1, "texto", true, null];  
console.log(JSON.stringify(numeros));  
// [1,"texto",true,null]
```

# Parámetro replacer

## Función personalizada

```
const usuario = {  
  nombre: "Carlos",  
  edad: 35,  
  password: "secreto",  
  email: "carlos@ejemplo.com"  
};  
  
const resultado = JSON.stringify(usuario, (clave, valor) => {  
  return clave === "password" ? undefined : valor;  
});  
  
console.log(resultado);  
// {"nombre":"Carlos","edad":35,"email":"carlos@ejemplo.com"}
```

## Array de claves permitidas

# JSON.parse() - Conversión desde JSON

## Sintaxis básica

`JSON.parse(cadena[, reviver])`

- ▶ cadena: Cadena JSON válida
- ▶ reviver (opcional): Función para transformar valores

## Ejemplo básico

```
'use strict';  
let cadena = '{ "nombre":"redes", "curso":1, "horario":["L1500", "X1700"] }';  
console.log(typeof cadena, cadena);  
// string {"nombre":"redes", "curso":1, "horario":["L1500", "X1700"] }  
  
let objeto = JSON.parse(cadena);  
console.log(typeof objeto, objeto);  
// object { nombre: 'redes', curso: 1, horario: [ 'L1500', 'X1700' ] }
```

# Ejemplos de JSON.parse()

## Objetos anidados

```
const jsonStr = `{
  "usuario": {
    "nombre": "María",
    "detalles": {
      "edad": 28,
      "ciudad": "Madrid"
    }
  }
}`;

const obj = JSON.parse(jsonStr);
console.log(obj.usuario.detalles.ciudad);
// Madrid
```

## Arrays complejos

## Con función reviver

```
const jsonStr = '{"fecha":"2024-01-15"}';

const obj = JSON.parse(jsonStr, (clave, valor) => {
  if (clave === "fecha") {
    return new Date(valor);
  }
  if (clave === "valor") {
    return parseInt(valor);
  }
  return valor;
});

console.log(obj.fecha.getFullYear());
console.log(typeof obj.valor);
```

# Manejo de Errores

## JSON.parse() con try-catch

```
function parsearJSONSeguro(jsonString) {  
  try {  
    return JSON.parse(jsonString);  
  } catch (error) {  
    console.error('Error parseando JSON:', error.message);  
    return null;  
  }  
}
```

*// Ejemplos de errores comunes*

```
const casosInvalidos = [  
  '{nombre: "Juan"}',      // Claves sin comillas  
  '{"nombre': 'Juan'}",    // Comillas simples  
  '{"nombre": "Juan",}',   // Coma final  
  '{"edad": undefined}'    // Valor no permitido
```

# Casos Especiales y Consideraciones

## Valores no serializables

```
const obj = {  
  fecha: new Date(),  
  funcion: () => console.log("hola"),  
  indefinido: undefined,  
  nan: NaN,  
  infinito: Infinity  
};
```

```
console.log(JSON.stringify(obj));  
// {"fecha":"2024-01-15T10:30:00.000Z","nan":null,"infinito":null}
```

## Circular References

```
const obj = { nombre: "A" };  
obj.self = obj; // Referencia circular
```

## toJSON() personalizado

```
const usuario = {  
  nombre: "Laura",  
  edad: 29,  
  toJSON: function() {  
    return {  
      nombre: this.nombre,  
      mayorEdad: this.edad >= 18  
    };  
  }  
};
```

```
console.log(JSON.stringify(usuario));  
// {"nombre":"Laura","mayorEdad":true}
```

## Números especiales

# Ejemplos Prácticos Completos

## Almacenamiento y recuperación

*// Guardar datos en localStorage*

```
const configuracion = {  
  tema: "oscuro",  
  notificaciones: true,  
  idioma: "es",  
  ultimoAcceso: new Date()  
};
```

*// Guardar*

```
localStorage.setItem('config', JSON.stringify(configuracion));
```

*// Recuperar*

```
const configGuardada = JSON.parse(  
  localStorage.getItem('config'),  
  (clave, valor) => {
```

# Ejemplo: API REST Simulada

## Simulación de comunicación con servidor

*// Simular respuesta de API*

```
function simularAPI() {  
  const respuestaAPI = JSON.stringify({  
    status: "success",  
    data: {  
      usuarios: [  
        { id: 1, nombre: "Ana", activo: true },  
        { id: 2, nombre: "Luis", activo: false }  
      ],  
      paginacion: {  
        pagina: 1,  
        total: 2  
      }  
    },  
    timestamp: new Date().toISOString()  
  })  
}
```

# Mejores Prácticas

## Recomendaciones generales

- ▶ **Siempre usar try-catch** con `JSON.parse()`
- ▶ **Validar el formato** antes de procesar
- ▶ **Usar replacer/reviver** para transformaciones complejas
- ▶ **Manejar referencias circulares** en objetos complejos
- ▶ **Considerar el rendimiento** con JSON muy grandes

## Función de validación robusta

```
function esJSONValido(str) {  
    try {  
        JSON.parse(str);  
        return true;  
    } catch {  
        return false;  
    }  
}
```