UD 01.03 JSON en Java

- Introducción a JSON
- APIs Java para JSON
  - *GSON*
  - *Jackson*
  - *Boon*
  - *JSON.org*
  - *JSONP*
- Implementación de un parser (analizador) JSON propio.

JSON es la abreviatura de **JavaScript Object Notation**. JSON es un formato de **intercambio de datos popular entre navegadores y servidores web porque los navegadores pueden analizar JSON en objetos JavaScript de forma nativa**. En el servidor, sin embargo, **JSON debe analizarse y generarse mediante las API de JSON**. Este apartado estudiaremos las diversas opciones que tiene para Java analizar y generar JSON.

JSON (JavaScript Object Notation) es un formato de datos independiente del lenguaje que expresa objetos JSON como listas legibles por humanos de propiedades (pares de nombre/valor).

> *Nota: JSON permite que el separador de línea Unicode U+2028 y el separador de párrafo U+2029 aparezcan sin escapar en cadenas entre comillas. Dado que JavaScript no admite esta característica,* ***JSON no es un subconjunto adecuado de JavaScript****.*

JSON se utiliza normalmente para la **comunicación asincrónica entre el navegador y el servidor a través de AJAX (Ajax)**.

También se utiliza con **sistemas de gestión de bases de datos NoSQL como MongoDb y CouchDb**; en aplicaciones de sitios web de **redes sociales como Twitter, Facebook, LinkedIn y Flickr; e incluso con la API de Google Maps.**

> *Nota: Muchos desarrolladores prefieren JSON sobre XML porque consideran que* ***JSON es menos extenso y más fácil de leer****. Consulta "JSON: la alternativa baja en calorías a XML" JSON: The Fat-Free Alternative to XML para obtener más información.*

Veremos cuáles son las **API JSON que existen en Java (no están incluidas en JDK)**, así como trabajar con archivos JSON en Java en general.

Última actualización: 12.10.2023 Autor/a: Pepe Calo 

01.00 INTRODUCCIÓN A JSON

1. ¿Qué es JSON?

JSON significa: **J**ava**S**cript **O**bject **N**otation (**Notación de Objetos de JavaScript**).

Es un formato para estructurar datos. Este formato es **utilizado por diferentes aplicaciones web para comunicarse entre sí**.

JSON es un formato de **intercambio de datos popular entre navegadores y servidores web porque los navegadores pueden analizar JSON en objetos JavaScript de forma nativa**.

En el servidor, sin embargo, **JSON debe analizarse y generarse mediante las API de JSON**.

JSON es un formato de datos **independiente del lenguaje** que expresa objetos JSON como listas legibles por humanos de propiedades (pares de nombre/valor).

> *Nota: JSON permite que el separador de línea Unicode U+2028 y el separador de párrafo U+2029 aparezcan sin escapar en cadenas entre comillas. Dado que JavaScript no admite esta característica,* ***JSON no es un subconjunto adecuado de JavaScript****.*

JSON se utiliza normalmente para la **comunicación asincrónica entre el navegador y el servidor a través de AJAX (Ajax)**.

También se utiliza:

- En **Sistemas de gestión de bases de datos NoSQL como MongoDb y CouchDb**.
- En aplicaciones de sitios web de **redes sociales como Twitter, Facebook, LinkedIn y Flickr**
- Incluso con la **API de Google Maps.**

Podría decirse que es el **sustituto del formato de intercambio de datos XML**:

- Es **fácil estructurar los datos** en comparación con XML.
- Admite **estructuras de datos** como arrays y objetos.
- Los documentos JSON se **ejecutan rápidamente** en el servidor o en cualquier lenguaje que disponga de biblioteca correspondiente.

> *La sintaxis de JSON procede de la notación de objetos de JavaScript, pero* ***el formato de JSON es sólo texto****. La generación y lectura de JSON existe para muchos lenguajes, que suelen disponer de bibliotecas para hacerlo.*
>
> *Nota: Muchos desarrolladores prefieren JSON sobre XML porque consideran que JSON es menos extenso y más fácil de leer. Consulta "JSON: la alternativa baja en calorías a XML" (JSON: The Fat-Free Alternative to XML para obtener más información.*

Veremos cuales son las **API JSON existen en Java (no están incluidas en JDK)**, así como trabajar con archivos JSON en Java en general.

2. Características

- Es un formato **independiente del lenguaje** que se deriva de JavaScript.
- Es **legible** y ***escribible*** por humanos, ya que es un formato de texto plano utilizando la notación de objetos de JavaScript.
- Es un formato de intercambio de datos **basado en texto y ligero**, lo que significa que es más sencillo de leer y escribir en comparación con XML.

- Aunque se deriva de un subconjunto de JavaScript, es **independiente del lenguaje**. Por lo tanto, el código para generar y analizar datos JSON se puede escribir en cualquier otro lenguaje de programación, como Java.
- **Transmisión de Datos entre Computadoras:** JSON se utiliza para enviar datos entre computadoras y programas.

3. Reglas sintácticas

Los datos están organizados en **pares de nombre/valor separados por comas**. Utiliza llaves para contener **los objetos { } y corchetes [ ] para contener los arrays**.

JSON presenta un **objeto JSON** como una lista delimitada por llaves y separada por comas de propiedades (**una coma no aparece después de la última propiedad**):

```
{
  propiedad1,
  propiedad2,
  ...
  propiedadN
}
```

Para cada propiedad, el **nombre se expresa como una cadena** que generalmente está **entre comillas dobles**. La cadena del nombre **se sigue por dos puntos**, que a su vez es seguido por un valor de un tipo específico. Ejemplos incluyen `"nombre": "Otto"` y `"edad": 4`.

JSON admite los siguientes seis tipos, que veremos más adelante:

- **Cadena:** una secuencia de cero o más caracteres Unicode. Las cadenas están **delimitadas por comillas dobles y admiten una sintaxis de escape con barra invertida**.
- **Número:** un número decimal (en base 10) que **puede contener una parte fraccional y puede usar notación exponencial (E)**.
- **Booleano:** Cualquiera de los valores `true` o `false`.
- **Array:** una lista ordenada de **cero o más valores**, cada uno de los cuales puede ser de cualquier tipo. Los arrays utilizan la **notación de corchetes cuadrados con elementos separados por comas**.
- **Objeto:** una colección **no ordenada** de propiedades donde los nombres (también llamados claves) son cadenas. Dado que los objetos están destinados a representar arrays asociativos, **se recomienda**, aunque no es obligatorio, que **cada clave sea única dentro de un objeto**. Los objetos están delimitados por llaves y usan comas para separar cada propiedad. Dentro de cada propiedad, los dos puntos separan la clave de su valor.
- **Nulo:** Un valor vacío, utilizando la palabra clave `null`.

Ejemplo

```
{
    "Libros": [
        {
            "Nombre": "Árboles",
            "Curso": "Introducción a los árboles",
            "Contenido": ["Árbol Binario", "BST", "Árbol Genérico"]
        },
        {
            "Nombre": "Grafos",
            "Temas": ["BFS", "DFS", "Orden Topológico"]
        }
    ]
}
```

3.1. Sintaxis JSON y reglas

La sintaxis JSON es un subconjunto de la sintaxis de JavaScript.

La sintaxis JSON se deriva de la sintaxis de la notación de objetos de JavaScript:

- Los datos están en **pares de nombre/valor**.
- Los **datos están separados por comas**.
- Las **llaves** (`{}`) contienen **objetos**.
- Los **corchetes** (`[]`) contienen **arrays**.

3.2. Datos JSON - “clave”: valor

Los datos JSON se escriben como pares de nombre/valor (también conocidos como pares clave/valor).

Un par de nombre/valor consiste en un **nombre de campo (entre comillas dobles), seguido de dos puntos y luego un valor**.

Ejemplo

```
"nombre": "Otto"
```

Los nombres JSON requieren comillas dobles.

3.2. JSON - se evalúa como objetos de JavaScript

El formato JSON es casi idéntico a los objetos de JavaScript.

En JSON, **las claves deben ser cadenas, escritas entre comillas dobles**.

JSON:

```
{"nombre": "Otto"}
```

4. Ventajas de JSON

- Almacena **todos los datos en un array** para que la transferencia de datos sea más fácil. Es la mejor opción para compartir datos de cualquier tamaño, incluso audio, video, etc.
- Su **sintaxis es muy pequeña, fácil y liviana**, por lo que ejecuta y responde de manera más rápida.
- Tiene un amplio rango de **compatibilidad con el navegador y es compatible con los sistemas operativos**. No requiere mucho esfuerzo para hacerlo compatible con todos los navegadores.
- En el lado del servidor, el análisis es la parte más importante que los desarrolladores desean. Si el **análisis es rápido** en el lado del servidor, el usuario puede obtener una respuesta rápida, por lo que en este caso, el análisis del lado del servidor de JSON es un punto fuerte en comparación con otros.

5. Desventajas de JSON

- La principal desventaja es que **no hay manejo y gestión de errores**. Si hay un pequeño error en el script, no se obtendrán datos estructurados.
- Se vuelve bastante peligroso cuando se usa con algunos **navegadores no autorizados**. Como el servicio JSON devuelve un archivo JSON envuelto en una llamada a función que debe ser ejecutada por los navegadores, si los navegadores no están autorizados, **tus/los datos pueden ser hackeados**.
- Tiene herramientas con soporte limitado que podemos usar durante el desarrollo.

6. Tipos de datos JSON

JSON (JavaScript Object Notation) es el formato de datos más ampliamente utilizado para el intercambio de datos en la web. JSON es un **formato de intercambio de datos basado en texto y completamente independiente del lenguaje**. Se basa en un subconjunto del lenguaje de programación JavaScript y es fácil de entender y generar.

6.1. Tipos de datos JSON

En JSON, los valores deben ser uno de los siguientes tipos de datos:

- Una cadena (string)
- Un número (number)
- Un objeto (object)
- Un array (array)
- Un booleano (boolean)
- `null`

> *A diferencia, en JavaScript, los valores pueden ser todos los anteriores, además de cualquier otra expresión JavaScript válida, incluyendo:*
>
> - Una función (function)
> - Una fecha (date)
> - `undefined`

JSON admite principalmente 6 tipos de datos:

String (Cadena)

Las cadenas JSON deben escribirse **entre comillas dobles, al igual que en el lenguaje Java o C**.

En JSON, los valores de tipo cadena deben escribirse entre comillas dobles:

Ejemplo

```
{"nome":"Wittgenstein"}
```

Hay varios caracteres especiales (**caracteres de escape**) en JSON que se pueden usar en cadenas, como **\ (barra invertida), / (barra diagonal), b (retroceso), n (nueva línea), r (retorno de carro), t (tabulación horizontal), etc**.

Ejemplo:

```
{ "poeta":"Sylvia Plath" }
{ "obra":"Ariel\/Sirenita", "género": "Poesía" }
```

Aquí / se utiliza como caracter de escape para / (barra diagonal).

Number (Número)

Se representa en base 10 y **no se utilizan formatos octales ni hexadecimales**.

Un número decimal firmado que **puede contener una parte fraccional y puede usar notación exponencial** (E).

> *JSON no permite NotANumber (como NaN),* ***no hace distinción entre enteros y punto flotante****. Además, como he comentado anteriormente* ***JSON no reconoce los formatos octal y hexadecimal****. (Aunque JavaScript utiliza un formato de punto flotante de doble precisión para todos los valores numéricos, otros lenguajes que implementan JSON pueden codificar los números de manera diferente).*

Ejemplo:

```
{ "edad": 32 }
{ "calificación": 9.5 }
```

Boolean (Booleano)

Este tipo de datos puede ser verdadero (`true`) o falso (`false`).

Ejemplo:

```
{ "premioPulitzer": true }
```

Null (Nulo)

Es simplemente un valor nulo definido.

Ejemplo

```
{
  "premioNobel": null,
  "publicaciones": 25
}
```

Object (Objeto)

Es un conjunto de **pares de nombre o valor insertados entre {} (llaves)**. Las claves deben ser cadenas y deben ser únicas. Múltiples pares de claves y valores se separan por una coma (,).

Dado que los objetos están destinados a **representar arrays asociativos**, se recomienda, aunque no es obligatorio, que **cada clave sea única dentro de un objeto**. Los objetos están delimitados por llaves y usan comas para separar cada propiedad. Dentro de cada propiedad, los dos puntos separan la clave de su valor.

Sintaxis:

```
{ "clave" : valor, .......}
```

Ejemplo:

```
{
  "Poeta": {
    "nombre": "Sylvia Plath",
    "edad": 32,
    "géneroLiterario": "Poesía"
  }
}
```

Array

Es una colección ordenada de **cero o más valores** y **comienza con [(corchete izquierdo) y termina con]** (corchete derecho). Los valores del array están **separados por ,** (coma).

Sintaxis:

```
[ valor, .......]
```

Ejemplo:

```
{
  "obras": ["Ariel", "The Bell Jar", "Colossus"]
}
```

```
{
  "colección": [
    {"añoPublicacion": 1965},
    {"añoPublicacion": 1971},
    {"añoPublicacion": 1960}
  ]
}
```

6.2. Archivos JSON

El **tipo de archivo para archivos JSON** es "`.json`". El tipo MIME para texto JSON es "**a** `pplication/json`".

7. Ejemplo completo de documento JSON

```
{
  "Poetas": [
    {
      "nombrePoeta": "Sylvia Plath",
      "obraDestacada": "Ariel",
      "géneroLiterario": "Poesía"
    },
    {
      "nombrePoeta": "Emily Dickinson",
      "obraDestacada": "The Collected Poems",
      "géneroLiterario": "Poesía"
    },
    {
      "nombrePoeta": "Walt Whitman",
      "obraDestacada": "Leaves of Grass",
      "géneroLiterario": "Poesía"
    }
  ]
}
```

› Ejercicio: Clasificación de la Liga ACB de Baloncesto

Última actualización: 12.10.2023 Autor/a: Pepe Calo 

01.01 JSON CON EL API JAVASCRIPT DE JAVA

1. Ejemplo de JSON con el API de Java (Scripting API)

En teoría, JSON no está en la API estándar de Java. Sin embargo, **podremos hacerlo con Java's Scripting API**.

> ***Nota**: En 2014, Oracle presentó una Propuesta de Mejora de Java (JEP) para agregar una API de JSON a Java. Aunque "JEP 198: Light-Weight JSON API" (http://openjdk.java.net/jeps/198) se actualizó en 2017, probablemente pasarán varios años antes de que esta API de JSON se convierta en parte de Java.*

En el siguiente ejemplo, sólo a modo de muestra, **podemos usar JavaScript, pero en un contexto de Java mediante la API de Scripting de Java**. (No te preocupes, no será demandado, pero es importante saber que existe). El siguiente código fuente Java permite ejecutar código JavaScript:

```
import java.io.FileReader;
import java.io.IOException;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
import static java.lang.System.*;

public class RunJSScript {
    public static void main(String[] args) {
        if (args.length != 1) {
            err.println("uso: java RunJSScript scriptEnJS");
            return;
        }
        ScriptEngineManager manager = new ScriptEngineManager(); // Inicio el API de Scripting
        ScriptEngine engine = manager.getEngineByName("nashorn");
        try {
            engine.eval(new FileReader(args[0])); // Sí, los flujos con importantes
        } catch (ScriptException se) {
            err.println(se.getMessage());
        } catch (IOException ioe) {
            err.println(ioe.getMessage());
        }
    }
}
```

Ojo: en versiones actuales de Java quizás debas añadir un motor de JavaScript a tu proyecto Maven, como ECMAScript como el proporcionado por Oracle GraalVM Oracle GraalVM for JDK 21

```
    <dependency>
        <groupId>org.graalvm.js</groupId>
        <artifactId>js</artifactId>
        <version>23.0.1</version>
    </dependency>
    <dependency>
        <groupId>org.graalvm.js</groupId>
        <artifactId>js-scriptengine</artifactId>
        <version>23.0.1</version>
```

```
    </dependency>
</dependency>
```

El método main anterior verifica primero que se haya especificado exactamente **un argumento desde línea de órdenes**, que es el nombre de un archivo de script. Si no es así, muestra información de uso y termina el programa. Por ello, debe recoger como argumento un programa/script en JavaScript, por ejemplo:

```
var poeta = {
    "nombre": "Sylvia",
    "apellidos": "Plath",
    "estaViva": false,
    "edad": 30,
    "direccion": {
        "direccionCalle": "21 2nd Street",
        "ciudad": "New York",
        "estado": "NY",
        "codigoPostal": "10021-3100"
    },
    "telefonos": [
        {
            "tipo": "casa",
            "numero": "212 555-1234"
        },
        {
            "tipo": "oficina",
            "numero": "646 555-4567"
        }
    ],
    "hijos": [],
    "marido": null
};
print(poeta.nombre);
print(poeta.apellidos);
print(poeta.direccion.ciudad);
print(poeta.telefonos[1].numero);
```

Explicación:

Suponiendo que se indicó un sólo argumento de línea de órdenes, se instancia la clase `javax.script.ScriptEngineManager`. `ScriptEngineManager` sirve como punto de entrada en la API de Scripting.

A continuación, se llama al método `ScriptEngine getEngineByName(String shortName)` del objeto `ScriptEngineManager` para **obtener un motor de script correspondiente al valor deseado** de `shortName`. Java 11 admite el motor de script `nashorn` (aunque ha sido obsoleto), que devuelve como un objeto cuya clase implementa la interfaz `javax.script.ScriptEngine`.

`ScriptEngine` declara **varios métodos** `eval()` **para evaluar un script**. `main()` invoca el método `Object eval(Reader reader)` para leer el script desde su objeto `java.io.FileReader` y (asumiendo que no se arroje `java.io.IOException`) luego evalúa el script. Este método devuelve cualquier valor de retorno del script, que ignoro. Además, este método arroja `javax.script.ScriptException` cuando ocurre un error en el script.

Compila:

```
javac RunJSScript.java
```

Suponiendo el Script se llama `poeta.js`, ejecuta la aplicación de la siguiente manera:

```
java RunJSScript poeta.js
```

Deberías observar la siguiente salida (junto con un mensaje de advertencia sobre la eliminación planeada de Nashorn en una futura versión de JDK):

```
Sylvia
Plath
New York
646 555-4567
```

Un objeto JSON existe como **texto independiente del lenguaje**. Para convertir el texto en un objeto dependiente del lenguaje, necesitas analizar el texto. **JavaScript proporciona un objeto `JSON` con un método `parse()` para esta tarea**. Pasa el texto a analizar como argumento a `parse()` y recibe el objeto basado en JavaScript resultante como el valor de retorno de este método. `parse()` lanza una `SyntaxError` cuando el texto no se ajusta al formato JSON.

Ejemplo de código JavaScript con `parse()`.

```
var tarjetaJSON = "{ \"numero\": \"1234567890123456\", " +
    "\"caducidad\": \"20/04\", \"tipo\": " +
    "\"visa\" }";
var tarjeta = JSON.parse(tarjetaJSON);
print(tarjeta.numero);
print(tarjeta.caducidad);
print(tarjeta.tipo);
var tarjetaJSON2 = "{ 'tipo': 'visa' }";
var tarjeta2 = JSON.parse(creditCardText2);
```

Suponiendo que el Script anterior se encuentra en `tarjeta.js`, ejecuta la aplicación de la siguiente manera:

```
java RunJSScript tarejta.js
```

Deberías observar la siguiente salida:

```
1234567890123456
20/04
visa
SyntaxError: JSON no válido: <json>:1:2 Se esperaba , o } pero se encontró '
{ 'type': 'visa' }
^ en <eval> en la línea número 11
```

El error de sintaxis muestra que **no puedes delimitar un nombre con comillas simples** (solo las comillas dobles son válidas).

› Ejercicio: lectura de datos de un archivo JSON con Java Script API

Última actualización: 12.10.2023 Autor/a: Pepe Calo

01.02 BIBLIOTECAS JSON PARA JAVA

1. Introducción

JSON es la abreviatura de **JavaScript Object Notation**. JSON es un formato de **intercambio de datos popular entre navegadores y servidores web porque los navegadores pueden analizar JSON en objetos JavaScript de forma nativa**. En el servidor, sin embargo, **JSON debe analizarse y generarse mediante las API de JSON**. Este apartado estudiaremos las diversas opciones que tiene para Java analizar y generar JSON.

JSON (JavaScript Object Notation) es un formato de datos independiente del lenguaje que expresa objetos JSON como listas legibles por humanos de propiedades (pares de nombre/valor).

> *Nota: JSON permite que el separador de línea Unicode U+2028 y el separador de párrafo U+2029 aparezcan sin escapar en cadenas entre comillas. Dado que JavaScript no admite esta característica,* ***JSON no es un subconjunto adecuado de JavaScript****.*

JSON se utiliza normalmente para la **comunicación asincrónica entre el navegador y el servidor a través de AJAX (Ajax)**.

También se utiliza con **sistemas de gestión de bases de datos NoSQL como MongoDb y CouchDb**; en aplicaciones de sitios web de **redes sociales como Twitter, Facebook, LinkedIn y Flickr; e incluso con la API de Google Maps.**

> *Nota: Muchos desarrolladores prefieren JSON sobre XML porque consideran que JSON es menos extenso y más fácil de leer. Consulta "JSON: la alternativa baja en calorías a XML" (JSON: The Fat-Free Alternative to XML para obtener más información.*

Veremos cuales son las **API JSON existen en Java (no están incluidas en JDK)**, así como trabajar con archivos JSON en Java en general.

2. APIs de JSON en Java

Cuando se popularizó el formato JSON, Java **no tenía una implementación estándar de analizador/generador JSON**. Por ello han surgido varias **implementaciones** de API de JSON de **código abierto para Java**.

Desde entonces, Java ha intentado abordar la API JSON de Java que falta en **JSR 353**, que no es un estándar oficial (de momento).

La comunidad Java también ha desarrollado varias API Java JSON de código abierto. Las **API JSON de Java de código abierto a menudo ofrecen más opciones y flexibilidad** en la forma en que puede trabajar con JSON que la API JSR 353. Por lo tanto, las API de código abierto siguen siendo opciones decentes (y mejores).

Algunas de las API Java JSON de código abierto más conocidas son:

1. GSON
    - https://github.com/google/gson
2. Jackson
3. mJson
4. Boon
5. JSON.org

> *También se puede utilizar **un analizador JSON compatible con JSR 353, JSONP**.*

Un ejemplo de rendimiento de las diferentes bibliotecas puede consultarse en el siguiente recurso:

https://github.com/fabienrenaud/java-json-benchmark#users-model

Hasta hace poco Jackson era el ganador, pero **en la actualidad GSON es, con diferencia, el más rápido**, seguido de cerca por JSONP, seguido por Jackson y luego *JSON.simple* en último lugar (no aparece Boon ni JSON.org en este análisis).

En la siguiente tabla se muestran ejemplos de los resultados porcentuales:

Velocidad de parsing	MB/ms	Tiempo de parsing
GSON	100%	0%
Jackson	58%	70.87%
JSON.simple	79%	126.58%
JSONP	44%	25.49%

GSON es un claro ganador, aunque con reservas.

1. GSON

GSON es una API Java JSON de Google. De ahí viene la G en GSON. GSON es razonablemente flexible, hasta hace poco, Jackson era más rápido que GSON. Pero hoy en día el rendimiento de GSON supera a otras alternativas:

https://github.com/google/gson

GSON contiene **3 analizadores Java JSON**:

- La clase Gson que puede analizar objetos JSON en objetos Java personalizados y viceversa, a traves de los métodos `fromJSon` y `toJson`, respectivamente.
- El **GSON JsonReader**, que es el analizador JSON de flujos de GSON, que **analiza un token JSON a la vez**.
- El **GSON JsonParser** que puede analizar JSON en una estructura de árbol de objetos Java específicos de GSON.

GSON también contiene un **generador JSON**:

- La clase**Gson** que puede**generar JSON a partir de clases Java personalizadas**.

2. Jackson

Jackson es una API Java JSON que proporciona varias formas diferentes de trabajar con JSON. Jackson es una de las API Java JSON más populares que existen. La página inicial de Jackson es la siguiente:

https://github.com/FasterXML/jackson

Jackson contiene dos **analizadores/parsers JSON** diferentes:

- El**Jackson ObjectMapper** que analiza JSON en objetos Java personalizados, o en una estructura de árbol específica de Jackson (modelo de árbol).
- El**Jackson JsonParser**, que es el analizador de extracción JSON de Jackson, analizando JSON un token a la vez.

Jackson también contiene dos **generadores JSON**:

- El**Jackson ObjectMapper** que puede generar JSON a partir de objetos Java personalizados, o de una estructura de árbol específica de Jackson (modelo de árbol).
- El**Jackson JsonGenerator** que puede generar JSON un token a la vez.

3. mJson

mJson es una pequeña biblioteca Java para JSON (creada por el desarrollador Borislav Lordanov) que se utiliza para analizar objetos JSON en objetos Java y viceversa. Esta biblioteca está documentada en GitHub (http://bolerio.github.io/mjson/, y presenta las siguientes características:

- Soporte **completo para la validación de JSON Schema Draft 4**.
- Un único tipo universal: todo es un objeto Json; no hay conversión de tipos.
- Un único método d**e tipo Factory para convertir un objeto Java en un objeto Json**; simplemente llama a `Json.make(cualquier objeto Java aquí)`.
- Análisis **rápido y codificado a mano**.
- Diseñado como una estructura de datos de propósito general para su uso en Java.
- Punteros de padre y método `up()` para recorrer la estructura JSON.
- Métodos concisos para leer (`Json.at()`), modificar (`Json.set()`, `Json.add()`), duplicar (`Json.dup()`), y fusionar (`Json.with()`).
- Fusión flexible de estructuras profundas (http://github.com/bolerio/mjson/wiki/Deep-Merging).
- - Métodos para la verificación de tipos (por ejemplo, `Json.isString()`) y acceso al valor subyacente de Java (por ejemplo, `Json.asString()`)
- Encadenamiento de métodos
- Factory adaptable para construir tu propio soporte para el mapeo arbitrario entre Java y JSON
- Biblioteca completa ubicada en un archivo Java, sin dependencias externas.

A diferencia de otras bibliotecas JSON, mJson **se centra en la manipulación de estructuras JSON en Java sin asignarlas a objetos Java fuertemente tipados**. Como resultado, mJson reduce la la escritura de código y permite trabajar con JSON en Java tan sencillo como en JavaScript.

4. Boon

Boon es una API Java JSON menos conocida, pero **supuestamente es la más rápida de todas** (según el último benchmark que he podido comprobar). Boon se está utilizando como la API JSON estándar en Groovy. Puedes encontrar a Boon aquí:

https://github.com/boonproject/boon

La API de **Boon es muy similar a la de Jackson** (por lo que es fácil de cambiar). Pero Boon es más que una API Java JSON. Boon es **un kit de herramientas de propósito general para trabajar con datos fácilmente**. Esto es útil, por ejemplo, dentro de los servicios REST, aplicaciones de procesamiento de archivos, etc.

Boon contiene los siguientes **analizadores Java JSON**:

- El**Boon ObjectMapper** que puede analizar JSON en objetos personalizados o mapas Java

Al igual que en Jackson, Boon ObjectMapper **también se puede utilizar para generar JSON a partir de objetos Java personalizados**.

5. JSON.org

JSON.org también tiene una API Java JSON de código abierto. Esta fue una de las primeras API Java JSON disponibles. Es razonablemente fácil de usar, pero no tan flexible o rápido como las otras API JSON mencionadas anteriormente. Puedes encontrar JSON.org aquí:

https://github.com/douglascrockford/JSON-java

Como también dice el repositorio de Github, esta es una **antigua API Java JSON.** No debe emplearse a menos que el proyecto ya lo esté usando. De lo contrario, buscca una de las otras opciones más actualizadas, preferiblemente GSON.

6. JSONP

JSONP es API JSON compatible con JSR 353 para Java. Ser **compatible con JSR 353 significa que si utiliza las API estándar, debería ser posible intercambiar la implementación de JSONP con otra API en el futuro, sin cambiar el código**. Puedes encontrar información JSONP en la página oficial:

https://jsonp.java.net/

Es de esperar que algunos proveedores de servidores de aplicaciones Java proporcionen API JSON compatibles con JSR 353 en el futuro (si no lo han hecho ya) puesto que es el formato de interdambio más empleado en la actualidad (por ejemplo para servicios REST).

Última actualización: 12.10.2023 Autor/a: Pepe Calo

01.03 GSON

1. Introducción

GSON es el **analizador (parser) y generador JSON de Google para Java**. Google desarrolló GSON para uso interno, pero lo abrió más tarde. GSON es razonablemente fácil de usar, pero quizás no tan sencillo como Jackson o Boon. En este apartado veremos cómo usar GSON para analizar objetos JSON en Java y serializar objetos Java en JSON.

GSON contiene varias API que se pueden usar para trabajar con JSON. Nos centraremos en los componente de GSON que analiza documentos JSON en en objetos Java o genera JSON a partir de objetos Java. Además del componente Gson, GSON también tiene un **analizador de extracción en el componente Gson JsonReader**.

Para utilizar GSON en su aplicación Java es necesario **incluir el archivo GSON JAR en la ruta de clases de su aplicación Java***. Puede hacerlo agregando GSON como una **dependencia de Maven a su proyecto**, o descargando el archivo JAR e inclúyalo en la ruta de clase manualmente:

```
<dependencies>
  <dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
      <version>2.10.1</version>
  </dependency>
</dependencies>
```

Enlaces:

- Archivos Jar de descarga disponibles en Maven Central.
- API Javadoc, documentacion de la versión más reciente.

2. Gson: convertir objetos Java a JSON y viceversa

Gson es una **biblioteca de Java que se puede utilizar para convertir objetos Java en su representación JSON**. También se puede utilizar para **convertir una cadena JSON en un objeto Java equivalente**.

Gson puede trabajar con objetos Java arbitrarios, incluidos los objetos preexistentes de los que no tiene el código fuente.

Existen algunos proyectos de código abierto que pueden convertir objetos Java a JSON, como los que hemos visto en el apartado anterior. Sin embargo, la **mayoría de ellos requieren que coloque anotaciones de Java en sus clases**, algo que no puede hacer si no tiene acceso al código fuente. Además, la mayoría de ellos no admiten

completamente el **uso de genéricos de Java**. Gson considera ambos como objetivos de diseño muy importantes y no precisa anotaciones y permite genéricos.

3. Características de Gson

- Proporciona métodos `toJson()` y `fromJson()` simples para **convertir objetos Java a JSON y viceversa**.
- Permite la **conversión de objetos preexistentes** y que no se puedan modificar a y desde JSON.
- Amplio **soporte de genéricos de Java**.
- Permite **representaciones personalizadas para objetos.**
- **Admite** objetos arbitrariamente complejos (con **jerarquías de herencia profundas y uso extensivo de tipos genéricos**).

4. Configuración y descarga

Dependiendo del tipo de proyecto empleado:

Gradle

```
dependencies {
  implementation 'com.google.code.gson:gson:2.10.1'
}
```

Maven

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.10.1</version>
</dependency>
```

Descarga del archivo JAR de GSON

Si el proyecto Java no emplea Maven, también se puede descargar el archivo JAR GSON directamente desde el repositorio central de Maven:

- http://search.maven.org
- Jar de Gson en Maven central

Una vez descargado el archivo JAR y puede agregarse al classpath de su aplicación Java:

- jar de Gson

Proceso de descarga de JAR y documentación

Gson se distribuye como un único archivo JAR; gson-2.10.1.jar es el archivo JAR más reciente, ahora. Para conseguir el archivo JAR, puedes ir al repositorio Maven este enlace, clic en el enlace de descargas y selecciona "jar" del menú desplegable, luego guarda el archivo gson-2.10.1.jar cuando se te pida hacerlo. Además, **es posible que desees descargar gson-2.10.1-javadoc.jar, que contiene la documentación de esta API**.

Nota: Gson tiene licencia según la Licencia Apache Versión 2.0 (www.apache.org/licenses/).

Es fácil trabajar con gson-2.10.1.jar. Simplemente **inclúyelo en el CLASSPATH al compilar el código fuente o al ejecutar una aplicación**, de la siguiente manera:

```
javac -cp gson-2.10.1.jar archivo_fuente
java -cp gson-2.10.1.jar;. archivo_clase_principal
```

5. Prerrequisitos

Versión mínima de Java

- Gson 2.9.0 y posterior: Java 7
- Gson 2.8.9 y anteriores: Java 6

A pesar de admitir versiones antiguas de Java, Gson también proporciona un **descriptor de módulo JPMS (nombre del módulo: `com.google.gson`) para usuarios de Java 9 o posterior**.

> ### Dependencias de JPMS (Java 9+)
>
> *Estos son los módulos opcionales del Sistema de Módulos de Plataforma Java (JPMS) en los que Gson depende. Esto sólo se aplica al ejecutar Java 9 o posterior.*
>
> - `java.sql` *(opcional desde Gson 2.8.9): Cuando este módulo está presente, Gson* ***proporciona adaptadores predeterminados para algunas clases de fecha y hora SQL****.*
> - `jdk.unsupported`*, respectivamente, la clase* `sun.misc.Unsafe` *(opcional): Cuando este módulo está presente, Gson* ***puede utilizar la clase*** `Unsafe` ***para crear instancias de clases sin constructor sin argumentos (sin constructor por defecto)****. Sin embargo, hay que tener cuidado al depender de esto.* `Unsafe` *no está disponible en todos los entornos y su uso tiene algunas trampas; consulta* `GsonBuilder.disableJdkUnsafe()`*.*

Nivel mínimo de API de Android

- Gson 2.11.0 y posterior: API nivel 21
- Gson 2.10.1 y anteriores: API nivel 19

Es posible que versiones antiguas de Gson también admitan niveles de API más bajos, aunque esto no se ha verificado.

¿Hay algo más en lo que pueda ayudarte?

6. Paquetes y clases Gson

Gson está compuesto por más de **30 clases e interfaces distribuidas en cuatro paquetes**:

https://www.javadoc.io/doc/com.google.code.gson/gson/latest/com.google.gson/module-summary.html

- **com.google.gson:** Este paquete proporciona acceso a Gson, la clase principal para trabajar con Gson.
- **com.google.gson.annotations:** Este paquete proporciona tipos de anotaciones para su uso con Gson.
- **com.google.gson.reflect:** Este paquete proporciona una clase de utilidad para obtener información de tipo de un tipo genérico.
- **com.google.gson.stream:** Este paquete proporciona clases de utilidad para leer y escribir valores codificados en JSON.

Empezaremos con la clase Gson, hablaremos de la deserialización de Gson (analizando objetos JSON), seguido por la serialización de Gson (creando objetos JSON). Terminaremos discutiendo brevemente características adicionales de Gson, como anotaciones y adaptadores de tipo.

Última actualización: 13.10.2023 Autor/a: Pepe Calo 

01.04 GSON. CREACIÓN DE INSTANCIAS GSON

1. Introducción a la Clase Gson

La clase Gson maneja la conversión entre JSON y objetos Java. Puedes instanciar esta clase utilizando el **constructor Gson()**, o puedes obtener una instancia de Gson trabajando con la clase **com.google.gson.GsonBuilder**. El siguiente fragmento de código demuestra ambos enfoques:

```
Gson gson1 = new Gson();
Gson gson2 = new GsonBuilder()
    .registerTypeAdapter(Id.class, new IdTypeAdapter())
    .serializeNulls()
    .setDateFormat(DateFormat.LONG)
    .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE)
    .setPrettyPrinting()
    .setVersion(1.0)
    .create();
```

Como norma general, **emplea Gson() cuando desees trabajar con la configuración predeterminada** (en la mayoría de los casos), y **utiliza GsonBuilder cuando quieras anular la configuración predeterminada**. Las llamadas a los métodos de configuración se encadenan, y el método `create()` de GsonBuilder se llama al final para devolver el objeto Gson resultante.

2. Creación de una instancia de Gson

Antes de poder usar GSON, primero debe **crearse un nuevo objeto Gson**. Hay dos formas de crear una instancia de Gson:

1. Usando el `new Gson()`
2. Crear una instancia de `GsonBuilder` e invocar al método create() en ella.

new Gson()

Puede crearse un objeto Gson simplemente creándolo con la orden: `new Gson();`. Así es como se ve la creación de un objeto Gson:

```
Gson gson = new Gson();
```

Una vez que haya creado una instancia de Gson, puede comenzar a **usarla para analizar y generar JSON**.

GsonBuilder.build()

Otra forma de crear una instancia de Gson es **crear un `GsonBuilder()` y llamar a su método `create()`**. Por ejemplo:

```
GsonBuilder builder = new GsonBuilder();
Gson gson = builder.create();
```

El uso de un GsonBuilder **permite establecer opciones de configuración en GsonBuilder antes de crear el objeto Gson**.

Gson admite la siguiente configuración predeterminada (la lista no está completa; consulta la documentación de Gson y GsonBuilder para obtener más información):

- **Gson proporciona serialización y deserialización predeterminadas** para instancias de `java.lang.Enum`, `java.util.Map`, `java.net.URL`, `java.net.URI`, `java.util.Locale`, `java.util.Date,` `java.math.BigDecimal` y `java.math.BigInteger`. Puedes cambiar la representación predeterminada registrando un adaptador de tipo (Lo veremos más adelante) a través de `GsonBuilder.registerTypeAdapter(Type, Object)`.
- El texto **JSON generado omite todos los campos nulos**. Sin embargo, **conserva los nulos en los arrays porque una matriz es una lista ordenada**. Además, si un campo no es nulo pero su texto JSON generado está vacío, se conserva el campo. Puedes configurar Gson para serializar valores nulos llamando a `GsonBuilder.serializeNulls()`.
- El formato de fecha predeterminado es el mismo que `java.text.DateFormat.DEFAULT`. Este formato ignora la porción de milisegundos de la fecha durante la serialización. Puedes cambiar el formato predeterminado invocando `GsonBuilder.setDateFormat(int)` o `GsonBuilder.setDateFormat(String)`.
- La política predeterminada de nombrado de campos para el texto JSON de salida es la misma que en Java. Por ejemplo, un campo de clase Java llamado `versionNumber` se mostrará como “versionNumber” en JSON. Las mismas reglas se aplican al mapear JSON entrante a clases Java. Puedes cambiar esta política llamando a `GsonBuilder.setFieldNamingPolicy(FieldNamingPolicy)`.
- El texto JSON generado por los métodos `toJson()` se representa de manera compacta: se eliminan todos los espacios en blanco innecesarios. Puedes cambiar este comportamiento llamando a `GsonBuilder.setPrettyPrinting()`.
- Por defecto, Gson ignora las anotaciones `@Since`. Puedes habilitar a Gson para que utilice estas anotaciones llamando a `GsonBuilder.setVersion(double)`.
- Por defecto, Gson ignora las anotaciones `@Expose`. Puedes habilitar a Gson para que serialice/deserialize solo aquellos campos marcados con esta anotación llamando a `GsonBuilder.excludeFieldsWithoutExposeAnnotation()`.
- Por defecto, Gson excluye campos transitorios o estáticos de la consideración para la serialización y deserialización. Puedes cambiar este comportamiento llamando a `GsonBuilder.excludeFieldsWithModifiers(int...)`.

Conversión entre primitivas JSON y sus equivalentes Java

Una vez que tienes un objeto Gson, **puedes llamar a varios métodos `fromJson()` y `toJson()` para convertir entre JSON y objetos Java**. Por ejemplo, código siguiente presenta una aplicación sencilla que obtiene un par de objetos Gson y demuestra la conversión entre JSON y objetos Java en términos de primitivas JSON.

```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import static java.lang.System.*;
```

```java
public class GsonDemo {
    public static void main(String[] args) {
        Gson gson = new Gson();

        // Deserialization
        String name = gson.fromJson("\"John Doe\"", String.class);
        out.println(name);

        // Serialization
        gson.toJson(256, out);
        out.println();
        gson.toJson("<html>", out);
        out.println();

        // Customized Gson with HTML escaping disabled
        gson = new GsonBuilder().disableHtmlEscaping().create();
        gson.toJson("<html>", out);
        out.println();
    }
}
```

El listado anterior declara una clase `GsonDemo` cuyo método `main()` primero instancia Gson, manteniendo su configuración predeterminada. Luego, invoca el método genérico `<T> T fromJson(String json, Class<T> classOfT)` de Gson para deserializar el texto JSON especificado (en json), basado en `java.lang.String`, en un objeto de la clase especificada (classOfT), que en este caso es String.

La cadena JSON “John Doe” (las comillas dobles son obligatorias), que se expresa como un objeto String de Java, se convierte (sin las comillas dobles) en un objeto String de Java. Una referencia a este objeto se asigna a `name`.

Después de imprimir el nombre devuelto, `main()` llama al método `void toJson(Object src, Appendable writer)` de Gson para convertir el entero autoboxeado 256 (almacenado por el compilador en un objeto `java.lang.Integer`) en un entero JSON y mostrar el resultado en la salida estándar.

`main()` vuelve a invocar `toJson()` para mostrar una cadena de Java que contiene `<html>`. Por defecto, Gson escapa los caracteres HTML `<` y `>`, por lo que estos caracteres no se imprimen. Para evitar este escape, es necesario obtener un objeto Gson a través de GsonBuilder, invocando el método `disableHtmlEscaping()` de GsonBuilder, que hace `main()` a continuación. Un segundo intento de imprimir `<html>` revela que no hay escape.

Última actualización: 13.10.2023 Autor/a: Pepe Calo