Serialización y manejo de archivos JSON

Serialización en Java. Escribir y leer objetos en archivos y JSONs.

Serialización en Java

La serialización es una herramienta que permite convertir cualquier objeto (que implemente la interfaz Serializable) en una secuencia de bytes que pueden ser posteriormente leídos para restaurar el objeto original. Esta característica se mantiene incluso a través de la red, por lo que podemos crear un objeto en una máquina que corre bajo el sistema operativo Windows, serializarlo y enviarlo a través de la red a una máquina que corre bajo UNIX donde será correctamente reconstruido.

La serialización es una característica agregada a Java que da soporte al **sistema RIM** (invocación de método remoto Java) y a la **persistencia.** El primero es un mecanismo que permite que un objeto en una máquina virtual Java invoque métodos en un objeto en otra máquina virtual Java. El segundo habla de, a través de la serialización, guardar el estado de un objeto para luego restaurarlo en un contexto distinto, ya sea, cambio de IDE, de lenguaje o por el mismo paso del tiempo. Además del estado del objeto, también se incluye la información de la clase.

Interfaz Serializable

Un objeto puede ser serializado si se implementa la interfaz Serializable. Si en una clase hay atributos que son de otras clases, éstos a su vez también deben implementar Serializable. Esta interfaz no declara ninguna función, se trata de una interfaz vacía:

public interface Serializable {
}

Para hacer una clase serializable simplemente debemos implementar esta interfaz:

public class Persona **implements** *Serializable* {

private String nombre;
…
}

Serial Version UID

El **UID** es el **Identificador Único de Versión Serial** asociado a una clase que se utiliza durante el proceso de serialización para garantizar que los objetos serializados y deserializados sean compatibles en diferentes versiones del mismo programa. Cuando se serializa un objeto en Java, se convierte en una secuencia de bytes que se puede transmitir o almacenar. Si se actualiza la definición de la clase del objeto y luego se intenta deserializar la versión antigua, pueden ocurrir problemas de compatibilidad. El Serial Version UID es una forma de resolver este problema: es un número de versión asociado con la clase que se utiliza para verificar si una clase serializada es compatible con la versión actual de la clase. Si no se declara el atributo serial, Java lo genera automáticamente (no es conveniente). Está de más decir, que en cada versión de la clase, el UID, debe cambiar para que la JVM detecte la incompatibilidad.

Se suele definir de la siguiente manera: **private static final long *serialVersionUID* = 8799656478674716638L;**

Modificador transient

Habrá ocasiones donde no será necesario incluir un atributo del objeto en la serialización, y para esto se encuentra el modificador ***transient***.

Este modificador le indica a la JVM que dicho atributo deberá ser exento de la serialización, en otras palabras, ignorará este atributo. Por otro lado, **los atributos que lleven el modificador static nunca se tomarán en cuenta al serializar el estado del objeto**, ya que este atributo pertenece a la clase y no al objeto. Los atributos estáticos se van a guardar como un estado de la clase, otorgando la posibilidad de revisar la compatibilidad de los UID. Al serializar el objeto, se serializa el estado de la clase de ese objeto, integrando atributos estáticos (útiles para el UID). Ejemplo del **transient**:

```
public class Persona implements Serializable {
        private String nombre;
        private transient String segundoNombre;
        … }
```

Serialización y archivos

Para poder transformar un objeto en un flujo de bits o recuperarlo de un archivo, se necesitan dos clases importantes pertenecientes a la API de serialización en Java:

ObjectOutputStream: Esta clase convierte el objeto en una secuencia de bytes que puede ser almacenada en un archivo o enviada a través de una red. Para usar esta clase, se debe crear un objeto ObjectOutputStream y llamar al método **writeObject()** para escribir el objeto en el flujo de salida.

ObjectInputStream: Esta clase lee los bytes del flujo de entrada y los convierte en un objeto. Para usar esta clase, se debe crear un objeto ObjectInputStream y llamar al método **readObject()** para leer el objeto desde el flujo de entrada.

Los constructores de ambos objetos deben recibir una instancia de las clases **FileOutputStream** y **FileInputStream** respectivamente. Se utilizan para escribir y leer bytes de datos en archivos. El constructor de estos objetos recibe como parámetro un objeto File. Por ejemplo, si quiero serializar un objeto voy a tener que transformarlo en un flujo de datos con ObjectOutputStream, pudiéndose guardar en un archivo gracias a FileOutputStream.

Serialización:

```
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(file));

oos.writeObject(new Persona("Juan", "Perez", 25));

oos.close();
```

Deserialización:

```
ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file));

Object aux = ois.readObject();

Persona persona = (Persona) aux;

System.out.println(persona);

ois.close();
```

Excepciones en serialización

Las clases mencionadas deben estar contenidas en un bloque try-catch porque pueden arrojar las siguientes excepciones:

- **IOException:** Esta es una excepción de nivel superior que indica que ha ocurrido un error durante la operación de entrada o salida. Puede ser arrojada por cualquier método de lectura o escritura de ObjectOutputStream o ObjectInputStream.
- **ClassNotFoundException:** Esta excepción es arrojada por ObjectInputStream cuando intenta leer un objeto de una clase que no se encuentra en el classpath del programa. Esto puede ocurrir si el objeto se creó con una versión anterior de la clase o si la clase se ha eliminado o renombrado.
- **NotSerializableException:** Esta excepción se arroja cuando intentas serializar un objeto que no implementa la interfaz Serializable. Es decir, si el objeto no puede ser convertido en una secuencia de bytes para ser almacenado o transmitido.
- **InvalidClassException:** Esta excepción se arroja cuando la versión de la clase de un objeto serializado no coincide con la versión de la clase que se está utilizando para deserializarlo.

JSON

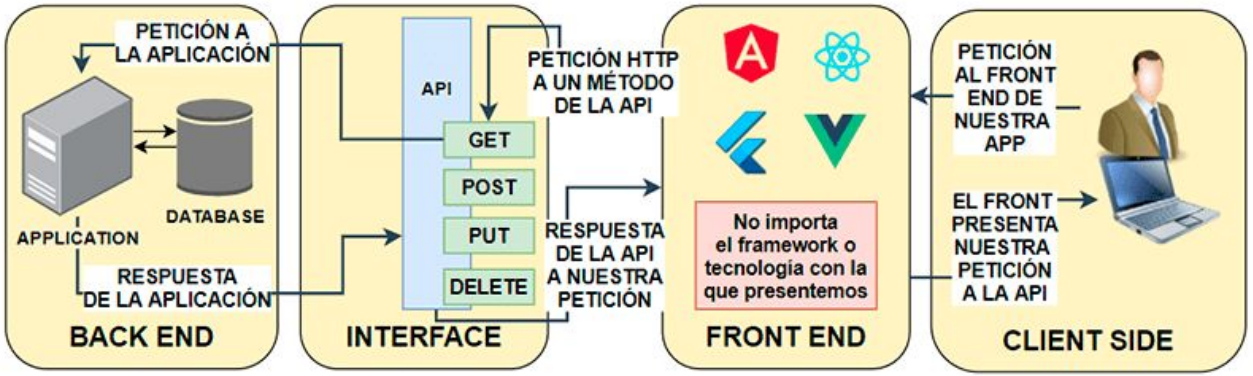
JSON (JavaScript Object Notation) es un formato de intercambio de datos ligero y de fácil lectura (tanto para los humanos como para las máquinas) que se utiliza para enviar y recibir datos entre distintos contextos (aplicaciones, sistemas, etc). JSON utiliza una sintaxis de llave-valor para representar los datos, separados por comas y encerrados entre { llaves }. Puede representar dos tipos estructurados: **Objetos y Matrices.**

Los objetos, al ser una notación de JavaScript, se escriben en formato clave-valor, siendo los valores cadenas, números, booleanos, nulos, otro objeto o una matriz. Esta última se asemeja a un arreglo.

JSON se utiliza comúnmente en aplicaciones web y móviles para enviar y recibir datos entre el servidor y el cliente. Supongamos que tenemos hecho el backend con Java y debemos enviar o recibir información del frontend escrito en JavaScript. La manera en que se van a comunicar estos dos lenguajes es a través de una API que va a manejar los estados del objeto en formato JSON.

```json
{
  "string": "Hi",
  "number": 2.5,
  "boolean": true,
  "null": null,
  "object": { "name": "Kyle", "age": 24 },
  "array": ["Hello", 5, false, null, { "key": "value", "number": 6 }],
  "arrayOfObjects": [
    { "name": "Jerry", "age": 28 },
    { "name": "Sally", "age": 26 }
  ]
}

```



Librerías para procesar JSONs

Las dos librerías para transformar objetos a JSON y viceversa son:

Jackson: Es una librería de utilidad de Java que nos simplifica el trabajo de serializar (convertir un objeto Java en una cadena de texto con su representación JSON), y deserializar (convertir una cadena de texto con una representación de JSON de un objeto en un objeto real de Java) objetos JSON. Para ello usa básicamente la introspección de manera que si en el objeto JSON tenemos un atributo “name”, para la serialización buscará un método “getName()” y para la deserialización buscará un método “setName(String s)”.

Gson: El uso de esta librería se basa en el uso de una instancia de la clase Gson. Dicha instancia se puede crear de manera directa (new Gson()) para transformaciones sencillas o de forma más compleja con GsonBuilder para añadir distintos comportamientos.

Para serializar con Jackson se debe utilizar un objeto de tipo **ObjectMapper** que permite mapear (convertir) un objeto a un formato de datos JSON y viceversa. Para escribir se utiliza el método **writeValue()** y para leer el método **readValue().** El primero recibe como parámetros el objeto de tipo File que representa al archivo y el objeto a guardar. El segundo, para leer, recibe el objeto File y la clase del objeto a mapear.

```
File file = new File(“mi_archivo.json”);

ObjectMapper mapper = new ObjectMapper();

// Escribir en un JSON:

Persona persona = new Persona("Juan", "Perez", 25);

mapper.writeValue(file, persona);

// Leer un JSON:

Persona p = mapper.readValue(file, Persona.class);

System.out.println(p);
```

Para serializar con Gson se utilizan las clases **BufferedWriter** y **BufferedReader**. Luego se necesita una instancia de la clase Gson la cual posee dos métodos: **fromJson()** y **toJson()**. El primero se utiliza para leer el archivo, recibe como parámetros el bufferedReader y la clase del objeto a Mapear. El segundo método se utiliza para escribir en el archivo, recibiendo como parámetros el objeto, la clase y el bufferedWriter.

```
File file = new File(“mi_archivo.json”);

Gson gson = new Gson();

BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(file));
BufferedReader bufferedReader = new BufferedReader(new FileReader(file));

// Escribir un archivo:

Persona persona = new Persona("Juan", "Perez", 25);

gson.toJson(persona, Persona.class, bufferedWriter);

// Leer un archivo:

Persona p = gson.fromJson(bufferedReader, Persona.class);

System.out.println(p);
```