Anexo I.- Parsers. XML y JSON

1. Introducción

Actualmente vivimos en la llamada Sociedad de la información donde el intercambio de información entre aplicaciones y máquinas es un requisito imprescindible. Pero, a pesar de lo que pueda parecer, este proceso es complicado ya que el origen y el destino de los datos deben comprender cómo codificar y decodificar la información. Además se necesita una disponibilidad global en dispositivos muy distintos como tablets, smartphones y otros dispositivos portátiles, con procesadores y formas de interpretar los datos de formas muy diferentes.

Para representar esta información podemos hacerlo mediante datos binarios o texto:

- **Datos binarios**: cualquier dato codificado en un ordenador que no sea texto se considera binario. Por ejemplo: un vídeo, una canción, una imagen, una aplicación informática, … son ejemplos de datos binarios. La forma de codificar varía en cada caso, por ejemplo, en el caso de las imágenes, cada punto (píxel) se codifica utilizando su nivel de rojo, verde y azul (RGB). Así por ejemplo, el código 11111111 00000000 00000000, se correspondería con el rojo puro (tiene a tope los niveles de rojo y totalmente apagados los niveles de verde y azul).
- **Texto**: la forma habitual de codificar el texto en forma de dígitos binarios ha sido establecer una equivalencia para cada carácter del lenguaje. Así por ejemplo, en el sistema de codificación ASCII, el carácter A se codifica como 01000001 en binario (65 en decimal). El problema es que existen diferentes sistemas de codificación para representar texto y hasta hace poco no había una estandarización que pudiese albergar a todos los alfabetos.

A la hora de decidir debemos tener en cuenta las ventajas y los inconvenientes que tenemos en cada caso.

a) Ventajas de los datos binarios:

- Ocupan menos espacio que el texto, ya que optimizan mejor su codificación a binario (por ejemplo el número 213 ocupa un solo byte y no 3 como ocurriría si fuera texto).
- Son más rápidos de manipular por parte del ordenador (se parecen más al lenguaje de la máquina).
- Permiten el acceso directo a los datos. Los archivos de texto siempre se manejan de forma secuencial.
- Los datos no son fácilmente interpretables, lo que aporta cierta ocultación al contenido. El contenido de los archivos de texto es fácilmente interpretable.
- Los archivos binarios son ideales para almacenar contenido cifrado.

b) Ventajas del texto:

- Son directamente modificables, sin tener que acudir a software específico.
- Su manipulación es más sencilla que la de los archivos binarios.
- Los dispositivos de red y software cliente permiten el paso de archivos de texto ya que no son susceptibles de contener virus informáticos.

2. El texto como formato más versátil

Los archivos de texto solo son capaces de almacenar texto plano, es decir, texto sin ningún formato y con la aparición de Unicode se ha podido universalizar la representación del texto, de forma que sea interpretado por todas máquinas por igual.
Debido a esta universalidad, se pueden utilizar archivos de texto para que el propio texto sirva para almacenar otros datos, es decir, información que no es texto. Para ello dentro del archivo habrá contenido que no se interpretará como texto sin más, sino que habrá texto especial, marcado de una forma que permita darle otro significado. Es lo que se conoce como **metadatos**.

La aparición de los lenguajes de marcas ha permitido estructurar el contenido de un archivo de texto mediante una serie de marcas, permitiendo darle un significado concreto. Entre las representaciones de marcado más utilizadas actualmente para el intercambio de información tenemos los formatos XML y JSON.

a) XML

Se trata de un subconjunto de SGML ideado para mejorar el propio SGML y con él poder definir lenguajes de marcado mediante etiquetas, con una sintaxis estricta y validable a través de Definición de Tipo de Documento (DTD) y/o Esquema XML (XML Schema) . El propósito principal del lenguaje es compartir datos a través de diferentes sistemas. Veamos un ejemplo:

```
<usuario>
    <nombre>John</nombre>
    <apellidos>Doe</apellidos>
    <domicilio>
        <direccion>C/ Mayor 25</direccion>
        <localidad>Gandia</localidad>
        <cp>46700</cp>
    </domicilio>
    <telefonos>
        <telefono tipo="fijo">961 234 567</telefono>
        <telefono tipo="móvil">777 888 999</telefono>
    </telefonos>
</usuario>
```

b) JSON

JavaScript Object Notation es una notación de datos procedente del lenguaje Javascript estándar (concretamente ECMAScript de 1999). Actualmente compite claramente con XML para la exportación/importación de datos dada su versatilidad, ya que permite definir datos complejos, como arrays u objetos, elementos pertenecientes al mundo de la programación de aplicaciones.

El texto se divide en datos y metadatos. De modo que el símbolo de los dos puntos separa el metadato del dato. Por otro lado, los símbolos de llave y corchete permiten agrupar de diversas formas los datos. Veamos un ejemplo:

```
{
  "usuario": {
    "nombre": "John",
    "apellidos": "Doe",
    "domicilio": {
      "direccion": "C/ Mayor 25",
      "localidad": "Gandia",
      "cp": 46700
    },
    "telefonos": [
      {
        "tipo": "fijo",
        "numero": 961234567
      },
      {
        "tipo": "móvil",
        "numero": 777888999
      }
    ]
  }
}
```

Aunque los dos formatos son utilizados de forma extensiva en la actualidad, el formato JSON se está imponiendo al ser el formato utilizado en la mayoría de API Rest, ya que los archivos suelen ocupar menos espacio sin perder legibilidad.

3. Parsers

La definición más común de parser (analizador sintáctico) es "programa informático que analiza una cadena de símbolos de acuerdo con las reglas de una gramática formal". Existen multitud de librerías y utilidades que simplifican el proceso de parsear documentos XML y JSON. En primer lugar vamos a ver las que trae Java de forma nativa.

a) XML

Los documentos XML tienen una jerarquía de unidades llamadas nodos; DOM, es una forma de describir

los nodos y las relaciones entre ellos. El objetivo del parser será recorrer el DOM e ir leyendo los valores asociados a cada uno de los elementos.

En nuestro caso vamos a utilizar JDOM que viene integrado en el propio lenguaje Java. Veamos las clases que intervienen en el parseado de documentos:

- **DocumentBuilderFactory** clase que implementa el patrón de diseño Factory para obtener instancias de DocumentBuilder.
- **DocumentBuilder** esta clase permite parsear un archivo (InputStream) y obtener un Document.
- **Document** esta clase representa un documento XML ya parseado, es decir, contiene el DOM del documento.
- **Element** representa el elemento raíz del Document.
- **NodeList** representa una lista de nodos de un determinado elemento. El método getElementsByTagName(String tag) de la clase Element devuelve un NodeList.
- **Node** representa cada uno de los nodos de un NodeList. El método item(int pos) de la clase NodeList devuelve un Node.

Veamos un ejemplo, suponiendo que tenemos un documento con la siguiente estructura:

```
<countries>
    <country countryCode="AF" countryName="Afghanistan" population="29121286" capital="Kabul" isoAlpha3="AFG" />
    <country countryCode="AI" countryName="Anguilla" population="13254" capital="The Valley" isoAlpha3="AIA" />
    <country countryCode="AL" countryName="Albania" population="2986952" capital="Tirana" isoAlpha3="ALB" />
    <country countryCode="AM" countryName="Armenia" population="2968000" capital="Yerevan" isoAlpha3="ARM" />

  ....

    <country countryCode="ZW" countryName="Zimbabwe" population="11651858" capital="Harare" isoAlpha3="ZWE" />
</countries>
```

Para parsear el documento podríamos definir el siguiente método:

```
public static Country[] parse(String path) throws IOException,
ParserConfigurationException, SAXException {
    /* Inicializamos a null el array de países */
    Country[] countries = null;
    try (FileInputStream fis = new FileInputStream(path)) {
        /* Obtenemos el DocumentBuilderFactory necesario para parsear mediante DOM */
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        /* Obtenemos el DocumentBuilder necesario para parsear mediante DOM */
        DocumentBuilder builder = factory.newDocumentBuilder();
        /* Obtenemos el Document parseando mediante DOM */
        Document dom = builder.parse(fis);
        /* Eliminamos nodos de texto redundantes (como saltos de línea o espacios) */
        dom.getDocumentElement().normalize();
        /* Obtenemos la lista de nodos con el tag "country" */
        NodeList items = dom.getElementsByTagName("country");
        /* Array de countries con tamaño igual al número de nodos de tipo country */
        countries = new Country[items.getLength()];
```

```
        /* Recorremos cada uno de los nodos */
        for (int i = 0; i < items.getLength(); i++) {
            /* Obtenemos el nodo de la posición i */
            Node item = items.item(i);
            /* Obtenemos los atributos necesarios para construir cada objeto Country */
            String countryCode =
item.getAttributes().getNamedItem("countryCode").getNodeValue();
            String countryName =
item.getAttributes().getNamedItem("countryName").getNodeValue();
            String countryCapital =
item.getAttributes().getNamedItem("capital").getNodeValue();
            long countryPopulation =
Long.parseLong(item.getAttributes().getNamedItem("population").getNodeValue());
            String countryIso3 =
item.getAttributes().getNamedItem("isoAlpha3").getNodeValue();
            /* Creamos el objeto Country en la posición i del array */
            countries[i] = new Country(countryCode, countryName, countryPopulation,
countryCapital, countryIso3);
        }
    }
    return countries;
}
```

b) JSON

Los documentos JSON están escritos en notación JSON y los tipos de datos que dispone son:

- **Números**: Se permiten números negativos y opcionalmente pueden contener parte fraccional separada por puntos. Ejemplo: 123.456
- **Cadenas**: Representan secuencias de cero o más caracteres. Se ponen entre doble comilla y se permiten cadenas de escape. Ejemplo: "Hola"
- **Booleanos**: Representan valores booleanos y pueden tener dos valores: true y false
- **null**: Representan el valor nulo.
- **Array**: Representa una lista ordenada de cero o más valores los cuales pueden ser de cualquier tipo. Los valores se separan por comas y el vector se mete entre corchetes. Ejemplo ["juan","pedro","jacinto"]
- **Objetos**: Son colecciones no ordenadas de pares de la forma <nombre>:<valor> separados por comas y puestas entre llaves. El nombre tiene que ser una cadena y entre ellas. El valor puede ser de cualquier tipo. Ejemplo: {"departamento":8,"nombredepto":"Ventas","director": "juan rodriguez","empleados":[{"nombre":"Pedro","apellido":"Fernandez"}, {"nombre":"Jacinto","apellido":"Benavente"}]}

En nuestro caso vamos a utilizar la librería *org.json*

Para importar la librería desde Gradle:

```
dependencies {
    implementation("org.json:json:20250517")
}
```

Veamos las clases que intervienen en el parseado de documentos:

- **String** para cargar todo el texto del archivo en un String.
- **JSONTokener** esta clase que permite interpretar la sintaxis JSON. Utilizada por JSONObject y JSONArray.
- **JSONArray** esta clase permite la carga de Arrays desde un JSONTokener.
- **JSONObject** esta clase permite la carga de Objetos desde un JSONTokener o de un JSONArray. Dispone de métodos para obtener atributos de distintos tipos:
  - getString, getDouble, etc.

Veamos un ejemplo:


```
[
  {
    "id": 0,
    "codAsig": "AAD",
    "nomAsig": "Acceso a base de datos"
  },
  {
    "id": 1,
    "codAsig": "DDI",
    "nomAsig": "Desarrollo de interfaces"
  },
  {
    "id": 2,
    "codAsig": "PMDM",
    "nomAsig": "Programación multimedia y dispositivos móviles"
  },
  {
    "id": 3,
    "codAsig": "PSP",
    "nomAsig": "Progroamación de servicios y procesos"
  },
  {
    "id": 4,
    "codAsig": "SGE",
    "nomAsig": "Sistemas de gestión empresarial"
  },
  {
    "id": 5,
    "codAsig": "EIE",
    "nomAsig": "Empresa e iniciativa emprendedora"
  },
  {
    "id": 6,
    "codAsig": "ANG",
    "nomAsig": "Inglés técnico"
  }
```

```
]
```

Para parsear el siguiente archivo podríamos hacer algo similiar a:

```
public Asignatura[] parseAsignaturas() throws IOException {
    Asignatura[] asignaturas = null;
    StringBuilder sb = new StringBuilder();
    try (InputStream is = getClass().getResourceAsStream("/asignaturas.json");
         BufferedReader br = new BufferedReader(new InputStreamReader(is,
StandardCharsets.UTF_8))) {
        String line;nnn
        while ((line = br.readLine()) != null) {
            sb.append(line);
        }
        JSONTokener tokener = new JSONTokener(sb.toString());
        JSONArray jsonArray = new JSONArray(tokener);
        asignaturas = new Asignatura[jsonArray.length()];
        for (int i = 0; i < jsonArray.length(); i++) {
            JSONObject jsonObject = jsonArray.getJSONObject(i);
            String codAsig = jsonObject.getString("codAsig");
            String nomAsig = jsonObject.getString("nomAsig");
            asignaturas[i] = new Asignatura(codAsig, nomAsig);
        }
    }
    return asignaturas;
}
```

4. Serialización con GSON

GSON librería desarrollada por Google para la serialización de objetos Java. Se caracteriza por ser muy sencilla de utilizar a la vez que eficiente.

Para utilizar la librería GSON en nuestro proyecto Java, solo tenemos que añadir la siguiente línea el archivo build.gradle:

```
dependencies {
    implementation "com.google.code.gson:gson:2.13.2"
}
```

o esta otra si el archivo de gradle es build.gradle.kts:

```
dependencies {
    implementation("com.google.code.gson:gson:2.13.2")
}
```

En el momento de escribir estas líneas la última versión es la 2.13.2 pero esto puede variar, consultar **https://github.com/google/gson** para obtener los números de la última versión.

El uso de esta librería es realmente sencillo, una vez realizada la importación desde Gradle, debemos crear un objeto Gson que será el que nos va a permitir serializar/deserializar nuestros objetos.

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

Serialización

Para serializar un objeto (generar un String con la representación del objeto en formato JSON) tenemos el método sobrecargado toJson(). De las diferentes variantes las más interesantes son:

- `String toJson(Object src)` al que le pasamos como parámetro el objeto que queremos serializar y nos devuelve una cadena de caracteres con el objeto en formato JSON.
- `void toJson(Object src, Appendable writer)` al que le pasamos como primer parámetro el objeto que queremos serializar y como segundo parámetro un objeto que implemente la interfaz Appendable, como por ejemplo un FileWriter.

Ejemplo:

Suponiendo que tenemos la siguiente clase:

```
public class Alumno {

    private int nia;
    private String nombre;
    private String apellido1;
    private String apellido2;
    private Date fechaNacimiento;
    public Alumno(int nia, String nombre, String apellido1, String apellido2, Date
fechaNacimiento) {
        this.nia = nia;
        this.nombre = nombre;
        this.apellido1 = apellido1;
        this.apellido2 = apellido2;
        this.fechaNacimiento = fechaNacimiento;
    }
    /** Getters y Setters necesarios */
    ....
}
```

y tenemos un Array de 10 objetos de tipo Alumno, podríamos serializarlo de la siguiente forma:

```
Gson gson = new Gson();

String alumnosJson = gson.toJson(alumnos);
```

El contenido de la variable alumnosJson sería similar a:

[{"nia":986523,"nombre":"César","apellido1":"Moya","apellido2":"Rico","fechaNacimiento":"May 9, 2000, 6:30:12 AM"},
{"nia":986524,"nombre":"Jorge","apellido1":"Briones","apellido2":"Delgadillo","fechaNacimiento":"Nov 22, 2003, 12:51:09 AM"},
{"nia":986525,"nombre":"Isabel","apellido1":"Merino","apellido2":"Galindo","fechaNacimiento":"Apr 9, 2003, 11:33:02 AM"},
{"nia":986526,"nombre":"Lucas","apellido1":"Ortiz","apellido2":"Portillo","fechaNacimiento":"Jan 17, 1992, 1:25:59 PM"},
{"nia":986527,"nombre":"Víctor","apellido1":"Santana","apellido2":"Flórez","fechaNacimiento":"Dec 4, 2006, 11:59:03 AM"},
{"nia":986528,"nombre":"Yolanda","apellido1":"Delarosa","apellido2":"Almanza","fechaNacimiento":"Oct 28, 2007, 3:51:06 PM"},

{"nia":986529,"nombre":"Sergio","apellido1":"Pulido","apellido2":"Cepeda","fechaNacimiento":"Mar 24, 2001, 2:41:01 PM"},
{"nia":986530,"nombre":"Julio","apellido1":"Villanueva","apellido2":"Rodrígez","fechaNacimiento":"May 7, 2007, 4:46:24 AM"},
{"nia":986531,"nombre":"Eva","apellido1":"Anaya","apellido2":"Mena","fechaNacimiento":"Jan 24, 2006, 7:00:21 PM"},
{"nia":986532,"nombre":"Lucia","apellido1":"Macías","apellido2":"Duran","fechaNacimiento":"Sep 19, 2003, 3:06:45 PM"}]

Si quisiéramos serializarlo a un archivo podriamos guardar el String generado a un archivo o utilizar la variante del método toJson() que permite indicar un Appendable.

```
Gson gson = new Gson();

try (FileWriter fw = new FileWriter("alumnos.txt");) {
    gson.toJson(alumnos, fw);
} catch (JsonSyntaxException jse) {
    jse.printStackTrace();
} catch (JsonIOException jioe) {
    jioe.printStackTrace();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

Deserialización

Para deserializar un objeto (generar un objeto Java a partir de un String en formato JSON) tenemos el método sobrecargado fromJson(). De las diferentes variantes las más interesantes son:

- `<T> T fromJson(String json, Class<T> classOfT)` al que le pasamos como primer parámetro la cadena en formato JSON y como segundo parámetro la clase que representa al objeto que queremos cargar.
- `<T> T fromJson(Reader json, Class<T> classOfT)` al que le pasamos como primer parámetro el archivo (FileReader) que contiene el JSON y como segundo parámetro la clase que representa al objeto que queremos cargar.

Ejemplo:

Para deserializar el Array de Alumnos a partir de un String serializado en JSON previamente:

```
Gson gson = new Gson();

try {
    Alumno[] alumnos2 = gson.fromJson(alumnosJson, Alumno[].class);
} catch (JsonSyntaxException jse) {

    jse.printStackTrace();
}
```

Si quisiéramos deserializarlo desde un archivo podríamos leer el archivo a un String y pasarlo al método anterior o bien utilizar la variante de fromJson que permite pasar como parámetro un fichero (Reader).

```
try (FileReader fr = new FileReader("alumnos.txt")){

    Alumno[] alumnos2 = gson.fromJson(fr, Alumno[].class);
} catch (FileNotFoundException fnfe) {
```

```
    fnfe.printStackTrace();
} catch (JsonSyntaxException jse) {
    jse.printStackTrace();
} catch (JsonIOException jioe) {
    jioe.printStackTrace();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

Limitaciones

Como hemos visto, serializar/deserializar objetos Java utilizando la librería GSON es realmente sencillo, pero si queremos serializar Interfaces o jerarquías de clases con Polimorfismo deberemos realizar tarea extra para indicarle el tipo del objeto al serializar y así poder cargarlo en la clase correcta al deserializar.

Para ello debemos escribir una clase que se encargue de modificar el proceso de serialización/deserialización implementando las interfaces `JsonSerializer` y `JsonDeserializer`. Estas dos interfaces nos obligan a implementar los métodos:

- `JsonElement serialize(Object src, Type typeOfSrc, JsonSerializationContext context)`
- `Object deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context)`

Al serializar lo que haremos será añadir un atributo extra a los objetos que nosotros queramos (veremos más adelante como indicarlo) que nos permitirá indicar el tipo real de cada objeto que estamos serializando.

```
private static final String CLASSNAME = "CLASSNAME";
private static final String DATA = "DATA";

@Override
public JsonElement serialize(Object src, Type typeOfSrc, JsonSerializationContext
context) {
    JsonObject jsonObject = new JsonObject();
    jsonObject.addProperty(CLASSNAME, src.getClass().getName());
    jsonObject.add(DATA, context.serialize(src));
    return jsonObject;
}
```

De esta forma, para los objetos que registremos un adaptador se añadirá un atributo llamado CLASSNAME cuyo valor será el tipo real del objeto y otro atributo llamado DATA que contendrá los datos del objeto.

Para deserializar aplicaremos el proceso inverso, para aquellos objetos para los que hayamos registrado un adaptador leeremos el atributo CLASSNAME para determinar su tipo real y cargaremos los datos leyendo el atributo DATA.

```
@Override
public Object deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext
context) throws JsonParseException {
        JsonObject jsonObject = json.getAsJsonObject();
        JsonPrimitive prim = (JsonPrimitive) jsonObject.get(CLASSNAME);
        String className = prim.getAsString();
        Class clase = getObjectClass(className);
        return context.deserialize(jsonObject.get(DATA), clase);
}

/**
 * Obtiene la Clase a partir del String indicado como parámetro
 * @param className
 * @return Class
 */
public Class getObjectClass(String className) {
    try {
        return Class.forName(className);
    } catch (ClassNotFoundException e) {
        //e.printStackTrace();
        throw new JsonParseException(e.getMessage());
    }
}
```

Para utilizar la clase adaptadora que implementa las interfaces JsonSerializer y JsonDeserializer debemos crear el objeto el objeto Gson utilizando un GsonBuilder de la siguiente forma:

```
GsonBuilder builder = new GsonBuilder();
/** Ahora registramos las clases que queramos personalizar su serialización */
builder.registerTypeAdapter(Estrategia.class, new AdaptadorGSON());
builder.registerTypeAdapter(Jugador.class, new AdaptadorGSON());
/** Finalmente creamos el objeto Gson */
Gson gson = builder.create();
/** A partir de este momento utilizamos Gson de la forma habitual */
```