BOLETÍN 01.04. GSON JAVA. PARSER

JSON CON GSON. PARSER

Introducción

1

Muchos de los siguientes ejercicios trabajan con archivo JSON de APIs JSON públicas o abiertas de Internet.

Existen varios modos de acceder a recursos, archivos, de Internet:

- **Java IO.**
- **Java NIO**.
- **Bibliotecas externas como AsyncHttpClient** y **Apache Commons IO**.

Java IO

La API más básica que podemos usar para descargar un archivo es Java IO. Podemos utilizar la **clase URL para abrir una conexión** al archivo que queremos descargar.

Para leer de manera eficaz el archivo, podemos utilizar el método **openStream()** para obtener un **InputStream**:

```
BufferedInputStream in = new BufferedInputStream(
        new URL(FILE_URL).openStream())
```

o:

```
URL url = new URL(FILE_URL);
URLConnection conn = url.openConnection();

BufferedReaderBufferedReader br = new BufferedReader(
                        new InputStreamReader(conn.getInputStream()));
```

La ventaja de este método es que podemos pasarle parámetros (cookies, tokens, identificadores de sesión…) a la cabecera HTTP:

```
URL url = new URL(FILE_URL);
HttpURLConnection urlc = (HttpURLConnection) url.openConnection();
urlc.setInstanceFollowRedirects(true);
urlc.setRequestProperty("User-Agent", "");
urlc.connect();
```

```
BufferedReader in =
new BufferedReader(new InputStreamReader(urlc.getInputStream()));
```

Al leer desde un InputStream, se recomienda **encapsularlo en un BufferedInputStream para aumentar el rendimiento**.

2

Como hemos visto, la mejorar de rendimiento proviene del almacenamiento en búfer. Al leer un byte a la vez mediante el método read(), cada llamada al método implica una llamada al sistema al sistema de archivos subyacente. Cuando la JVM invoca la llamada al sistema read(), el contexto de ejecución del programa cambia de modo usuario a modo kernel y viceversa.

Este cambio de contexto es costoso desde una perspectiva de rendimiento. Al leer un gran número de bytes, el rendimiento de la aplicación será deficiente debido al gran número de cambios de contexto involucrados.

Para **escribir los bytes leídos** desde la URL en nuestro archivo local, utilizaremos el método write() de la clase FileOutputStream:

```
try (BufferedInputStream in = new BufferedInputStream(
        new URL(FILE_URL).openStream());
    FileOutputStream fileOutputStream =
                new FileOutputStream(FILE_NAME)) {
    byte dataBuffer[] = new byte[1024];
    int bytesRead;
    while ((bytesRead = in.read(dataBuffer, 0, 1024)) != -1) {
        fileOutputStream.write(dataBuffer, 0, bytesRead);
    }
} catch (IOException e) {
    // gestión de la excepción
}
```

Cuando se utiliza un **BufferedInputStream**, **el método read() leerá tantos bytes como hayamos establecido para el tamaño del búfer**. En nuestro ejemplo, ya estamos haciendo esto al leer bloques de 1024 bytes a la vez, por lo que BufferedInputStream no es necesario.

Para archivos JSON, como son archivos de texto, precisamos **convertir el flujo entreda (InputStream) en un Reader por medio de la clase *InputStreamReader***, pasando a leer línea a línea.

Muchos métodos de procesado de archivos JSON (fromJson, parse,…) tienen una versión sobrecargada que recoge un Reader además de un String.

Java NIO

En el caso anterior, bajamos a nivel de flujo pero, como hemos estudiado, **a partir de Java 7, se dispone de la clase Files que contiene métodos auxiliares para manejar operaciones de entrada/salida (IO)**.

Se puede utilizar el método Files.copy() para leer todos los bytes de un InputStream y copiarlos a un archivo local:

```
InputStream in = new URL(FILE_URL).openStream();
Files.copy(in, Paths.get(FILE_NAME),
                StandardCopyOption.REPLACE_EXISTING);
```

El ejemplo anterior funciona bien, pero puede mejorarse. El principal ¿inconveniente? es que los bytes se almacenan en búfer en la memoria.

Java NIO tiene métodos para transferir bytes directamente entre dos canales sin almacenamiento en búfer.

El paquete Java NIO ofrece la posibilidad de **transferir bytes entre dos canales sin almacenarlos en el espacio de memoria de la aplicación**.

Para leer el archivo desde nuestra URL, crearemos un ***ReadableByteChannel*** a partir del flujo de URL:

```
ReadableByteChannel readableByteChannel =
                Channels.newChannel(url.openStream());
```

Los bytes leídos del ***ReadableByteChannel*** se transferirán a un ***FileChannel*** correspondiente al archivo que se va a descargar:

```
FileOutputStream fileOutputStream = new FileOutputStream(FILE_NAME);
FileChannel fileChannel = fileOutputStream.getChannel();
```

Puede usarse el método ***transferFrom()*** de la clase ***ReadableByteChannel*** para descargar los bytes desde la URL dada a nuestro ***FileChannel***:

```
fileChannel.transferFrom(readableByteChannel,
                                        0, Long.MAX_VALUE);
```

Los **métodos transferTo() y transferFrom() son más eficientes que simplemente leer desde un flujo utilizando un búfer**. Dependiendo del sistema operativo subyacente, los datos **pueden transferirse directamente desde la caché del sistema de archivos a nuestro archivo sin copiar ningún byte en el espacio de memoria de la aplicación**.

4

En sistemas Linux y UNIX, estos métodos utilizan la técnica de copia cero que reduce el número de cambios de contexto entre el modo kernel y el modo usuario.

Ejercicio 1. Búsqueda de códigos postales.

Existen muchas API libres o de código abierto en Internet. Una de las más curiosas es la que devuelve la localización a la que pertenece un código postal:

https://www.zippopotam.us/

Que está disponible para muchos países, entre ellos España:

Estructrua: api.zippopotam.us/*codigoPis/codigoPostal*
Ejemplo: https://api.zippopotam.us/es/15705

Ciudad->Zip: api.zippopotam.us/codigoPais/estado/ciudad
Ejemplo: https://api.zippopotam.us/es/GA/Santiago%20De%20Compostela

El formato del JSON es el siguiente:

```
{
   "post code": "15705",
   "country": "Spain",
   "country abbreviation": "ES",
   "places": [
       {
           "place name": "Santiago de Compostela",
           "longitude": "-8.5459",
           "state": "Galicia",
           "state abbreviation": "GA",
           "latitude": "42.8782"
       }
   ]
}
```

a) Crea una las clases Java necesarias para conversión a archivos Json: **Lugar** y **CodigoPostal**.

Emplea **estándares de nombres y conversión de tipos e datos** (los números no deben representarse como cadenas de texto). Emplea nombres significativos en galego/castelán, como consideres.

Sobrescribe los métodos ***toString***, ***equals*** y ***hashCode***.

Ten en cuenta el que código postal puede hacer referencia a varios lugares y que un lugar sólo puede tener un único código postal.

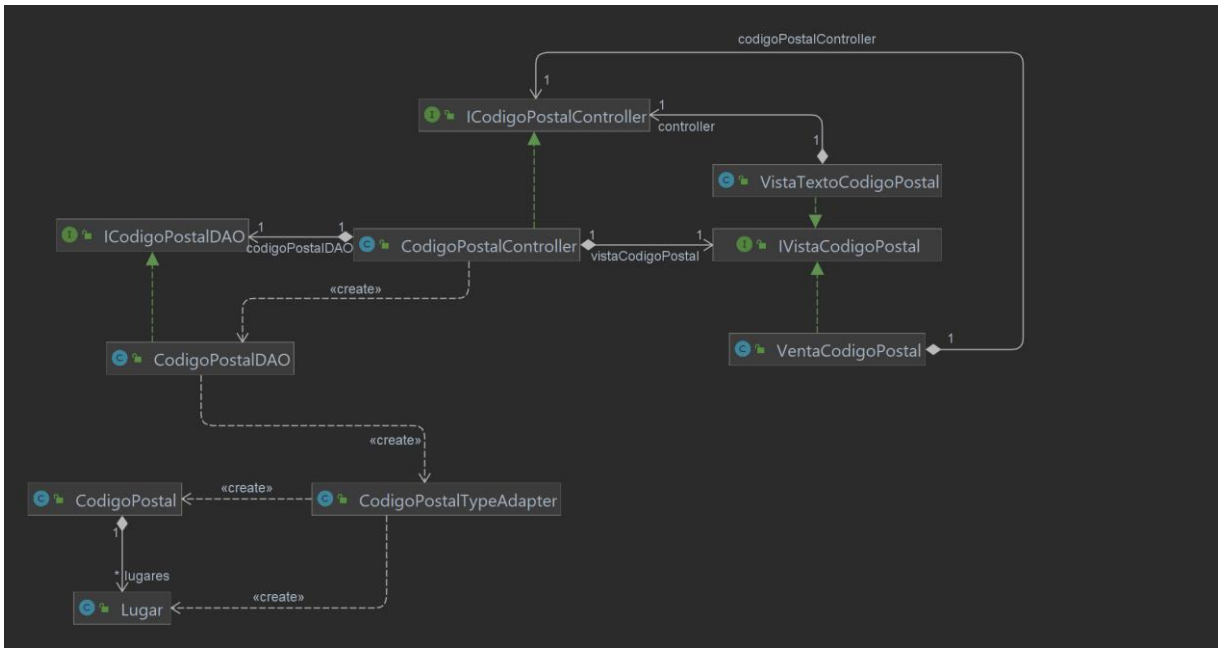
b) Crea una aplicación que, dado el código postal, muestre la lista de lugares que corresponden.

c) Haz lo mismo, pero de modo que **recoja la localidad (de Galicia o España) y muestre los códigos postales de la misma**. Inspecciona el JSON para tomar las decisiones de diseño que consideres oportunas.
https://api.zippopotam.us/es/an/ja%C3%A9n

Referencias de códigos de comunidades:
https://www.geonames.org/postalcode-search.html?q=&country=ES&adminCode1=M

d) Implementa la aplicación, sencilla, mediante el patrón de arquitectura modelo-vista-controlador de la aplicación, usando la vista que consideres: texto, ventana Swing, Android, JavaFX, Web…

Un ejemplo podría ser el siguiente diagrama:



Ejercicio 2. Joke API

Otra API sencilla es la Joke API, en la que puedes consultar entres 1369 chistes, aleatorio o por categoría, así como en varios idiomas:

6 https://sv443.net/jokeapi/v2/

El formato del JSON de salida es el siguiente:

```
{
    "error": false,
    "category": "Programming",
    "type": "twopart",
    "setup": "¿Por qué C consigue todas las chicas y Java no
tiene ninguna?",
    "delivery": "Porque C no las trata como objetos.",
    "flags": {
        "nsfw": false,
        "religious": false,
        "political": false,
        "racist": false,
        "sexist": false,
        "explicit": false
    },
    "safe": true,
    "id": 6,
    "lang": "es"
}
```

El formato de petición es el siguiente:

https://v2.jokeapi.dev/joke/Programming?lang=es

Las categorías son: ***Any*** (excluyente), ***Programming***, ***Miscellaneous***., ***Dark***, ***Pun***, ***Spooky***, ***Christmas***.

Además, se pueden solicitar **varias categorías** a la vez (menos Any):

https://v2.jokeapi.dev/joke/Programming,Dark,Christmas?lang=es

Las "banderas negras" (&**blacklistFlags**=nsfw) pueden ser: ***nsfw***, ***religious***, ***political***, ***racist***, ***sexist***, ***explicit***:

https://v2.jokeapi.dev/joke/Programming,Christmas?lang=es&blacklistFlags=nsfw

Se pide:

a) Crea las clases Java que consideres adecuada para la aplicación, empleando la nomenclatura estándar y guardando las banderas en una enumeración. Los atributos de las clases no tienen que ajustarse a los del archivo JSON.

b) Crea una clase ***ChisteDAO*** que obtenga los chistes del API. Al menos debe tener: ***getChiste()***, que devuelve uno aleatorio; ***getChisteByLang(String lang); getChisteByCategory(String category).***

c) Haz una aplicación con un menú que pida un tipo de chiste y lo muestre por pantalla. Si lo deseas, haz una aplicación gráfica empleando un patrón MVC.

Ejercicio 3. Juegos

Referencias: https://www.freetogame.com/api-doc

Disponemos de un archivo JSON los dos datos de un juego:

- Las **plataformas** pueden ser: **pc**, **browser**, **all**.
- Las **categorías** pueden ser:
  - mmorpg, shooter, strategy, moba, racing, sports, social, sandbox, open-world, survival, pvp, pve, pixel, voxel, zombie, turn-based, first-person, third-Person, top-down, tank, space, sailing, side-scroller, superhero, permadeath, card, battle-royale, mmo, mmofps, mmotps, 3d, 2d, anime, fantasy, sci-fi, fighting, action-rpg, action, military, martial-arts, flight, low-spec, tower-defense, horror, mmorts
- La **ordenación** puede ser: ***release-date***, ***popularity***, ***alphabetical*** o ***relevance***


```
{
    "id": 452,
    "title": "Call Of Duty: Warzone",
    "thumbnail": "https:\/\/www.freetogame.com\/g\/452\/thumbnail.jpg",
    "status": "Live",
     "short_description": "A standalone free-to-play battle royale and modes accessible via Call of Duty: Modern Warfare.",
      "description": "Call of Duty: Warzone is both a standalone free-to-play battle royale and modes accessible via Call of Duty: Modern Warfare. Warzone features two modes \u2014 the general 150-player battle royle, and \u201cPlunder\u201d. The latter mode is described as a \u201crace to deposit the most Cash\u201d. In both modes players can both earn and loot cash to be used when purchasing in-match equipment, field upgrades, and more. Both cash and XP are earned in a variety of ways, including completing contracts.\r\n\r\nAn interesting feature of the game is one that allows players who have been killed in a match to rejoin it by winning a 1v1 match against other felled players in the Gulag.\r\n\r\nOf course, being a battle royale, the game does offer a battle pass. The pass offers players new weapons, playable characters, Call of Duty points, blueprints, and more. Players can also earn plenty of new items by completing objectives offered with the pass.",
    "game_url": "https:\/\/www.freetogame.com\/open\/call-of-duty-warzone",
    "genre": "Shooter",
    "platform": "Windows",
```

```
    "publisher": "Activision",
    "developer": "Infinity Ward",
    "release_date": "2020-03-10",
    "freetogame_profile_url": "https:\/\/www.freetogame.com\/call-of-duty-warzone",
    "minimum_system_requirements": {
        "os": "Windows 7 64-Bit (SP1) or Windows 10 64-Bit",
        "processor": "Intel Core i3-4340 or AMD FX-6300",
        "memory": "8GB RAM",
        "graphics": "NVIDIA GeForce GTX 670 \/ GeForce GTX 1650 or Radeon HD 7950",
        "storage": "175GB HD space"
    },
    "screenshots": [
        {
            "id": 1124,
            "image": "https:\/\/www.freetogame.com\/g\/452\/Call-of-Duty-Warzone-1.jpg"
        },
        {
            "id": 1125,
            "image": "https:\/\/www.freetogame.com\/g\/452\/Call-of-Duty-Warzone-2.jpg"
        },
        {
            "id": 1126,
            "image": "https:\/\/www.freetogame.com\/g\/452\/Call-of-Duty-Warzone-3.jpg"
        },
        {
            "id": 1127,
            "image": "https:\/\/www.freetogame.com\/g\/452\/Call-of-Duty-Warzone-4.jpg"
        }
    ]
}
```


a) Crea las clases de la aplicación:

- ***Image***: con identificador, URL y ¡un array de bytes con la imagen!.
- ***Plataforma***: enumeración con 3 posibles valores, BROWSER, PC, ALL.
- ***Game***: con identificador, título, miniatura (tipo *Image*), descripción, url para jugar, género, plataforma (de tipo *Plataforma*), fecha de realización (*LocalDate*) y una lista de imágenes.
- ***BrowserGame***: hereda de Game y se trata de un juego para navegador, por lo que su categoría será BROWSER.

b) Haz una sencilla aplicación que, a partir de el JSON de un Game, cree un juego, pero sólo el identificador, el título, la descripción, la URL, ... sin miniatura ni la lista de imágenes.

c) Haz que el juego se pueda guardar en un archivo de texto con el nombre: *nombre del juego.txt* y la versión toString del Game dentro de él. Emplea Java IO, no Files.

d) Cómo sólo nos interesan los juegos de navegador para jugar en clase mientras Pepe explica, haremos una aplicación que descargue la lista de juegos de:

https://www.freetogame.com/api/games? platform=browser

Empleando un ***InstanceCreator*** para que asigne la plataforma BROWSER al constructor de *Game*.

e) Amplía el ejercicio anterior para que también recupere las imagenes, sin los bytes, sólo la url. La miniatura tendrá siempre id igual a 0.

f) Amplía el ejercicio apartado anterior para que guarde recupere también la imagen y la almacene en el array de bytes. Ved nota [1]

g) Usando el API haz una aplicación que pida un identificador de objeto y lo descargue, tanto en un fichero de texto como las imágenes. Previamente debe "deserializar el objeto en el tipo Game".

h) SI deseas hacer una aplicación gráfica, puedes ver la nota 2, en la que explico cómo crear un ImageIcon a partir de una array de bytes.

Nota 1. Guardar una imagen en un array de bytes

```
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class ImageToBytes {
    public static void main(String[] args) {
        try {
            // 1. Crea un objeto fis de tipo InputStream a la imagen.
            // Ya deberías saberlo
            // 2. Crea un flujo de salida a un array de bytes:
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            byte[] buf = new byte[1024]; // buffer
            for (int readNum; (readNum = fis.read(buf)) != -1;) {
                // Escribimos en el array de bytes
                bos.write(buf, 0, readNum);
            }
            // Convertimos el flujo de bytes en un array
            byte[] bytes = bos.toByteArray();
            System.out.println("Imagen convertida a bytes: " + bytes);
        } catch (IOException e) {
            // …
        }
    }
}
```

Nota 2. *ImageIcon* a partir de un array de bytes.

```
import java.io.FileOutputStream;
import java.io.IOException;

10  public class BytesToImageFile {
        public static void main(String[] args) {
            try {
                byte[] bytes = new byte[] { 0x00, 0x01, 0x02, …};
                 // Ya sabes que hay mejores maneras de crear flujos, más
                 // eficientes, pero a modo de ejemplo.
                FileOutputStream fos =
                           new FileOutputStream("ruta/a/tu/imagen.jpg");
                fos.write(bytes);
                fos.close(); // mejor, try-catch-with-resources.
                System.out.println("Imagen guardada en disco.");
            } catch (IOException e) {
                // ..
            }
        }
    }
```