UD1-Acceso a ficheros y RDBMS

CFGS DAM - MP Acceso a Datos

1. ¿QUÉ ES UN FICHERO?

Se considera un **fichero** (o archivo) en informática a un conjunto de datos que se encuentran almacenados en un dispositivo. Este conjunto de datos viene agrupado por un nombre, una ruta de acceso y una extensión que debe ser única.

Los ficheros pueden ser de **texto** o **binarios**. En los ficheros de texto, los bytes que contienen representan exclusivamente carácteres. Estos ficheros pueden crearse y visualizarse usando un editor de textos y se pueden leer/escribir carácter a carácter o por cadenas completas. En los ficheros binarios, los bytes existentes no representan exclusivamente caracteres, pudiendo ser números, objetos, imágenes, sonido…

Las operaciones que se pueden llevar a cabo con ficheros incluyen:

- Apertura: se "abre" el fichero para leerlo o escribirlo.
- Cierre: se "libera" el fichero.
- Lectura: se lee el fichero o una de sus partes.
- Escritura: se añaden datos o se sobrescriben los actuales.
- Ejecución: se usan los datos del fichero para ejecutar un software.
- Creación: se crea un nuevo fichero.
- Eliminación: se borra un fichero determinado

El tratamiento habitual de un fichero es el siguiente:

- **Apertura**: cuando se abre un fichero, este se reserva para operar con él. Se establece un flujo de datos desde el fichero a una variable, que representa al fichero.
- **Operaciones diversas**: a partir de esa variable, se pueden realizar todas las operaciones sobre el fichero que se deseen.
- **Cierre**: Cuando se quiere dejar de usar el fichero, se debe cerrar el mismo, cortando el flujo de datos y liberando la variable.

2. FICHEROS EN JAVA

2.1. Clase File

La clase que manipula los ficheros en Java se llama **File**. Permite hacer muchas

operaciones sobre un fichero y sus propiedades, pero **no permite leer ni escribir**.

El resto de clases que trabajan con ficheros necesitan objetos de esta clase, por lo que es la base de cualquier operación de manipulación de ficheros.

```
import java.io.File;

public class Ej21ConsultarFichero {
   public static void main(String[] args) {
       File fichero = new File("res/palabras.txt");
       if (fichero.exists()) {
           System.out.printf("Nombre del archivo:%s%n", fichero.getName());
           System.out.printf("Ruta %s", fichero.getPath());
           System.out.printf("Ruta absoluta %s%n",
fichero.getAbsolutePath());
           System.out.printf("Se puede escribir %s%n", fichero.canWrite());
           System.out.printf("Se puede leer %s%n", fichero.canRead());
           System.out.printf("Tamaño "+fichero.length());
       }
   }
}
```

2.2. Stream de datos

Para poder realizar las operaciones de lectura y escritura de ficheros, Java usa lo que se conoce como **Stream o flujo de datos**. Es una vía de comunicación entre programa y fichero que permite "moverse" por las distintas partes del fichero a través de un puntero que actúa como si fuera un cabezal de lectura/escritura. Los streams pueden ser de bytes o de carácteres.

Mientras que un archivo es una colección de datos almacenados en un disco con un nombre específico y una ruta de directorio, cuando se abre un archivo para su lectura o escritura, se convierte en un Stream(Flujo)**. El Stream es básicamente la secuencia de bytes que ocurren a través de la ruta de comunicación.** Hay dos principales flujos: el flujo de entrada y el flujo de salida. El flujo de entrada se utiliza para la lectura de los datos de archivo (operación de lectura) y el flujo de salida se utiliza para escribir en el archivo (operación de escritura).

2.3. Clase Path y Files en Java 7+

En Java 7 se introducen las clases Path y Files para la gestión de la entrada/salida (ficheros). La clase Path es una alternativa a la clase File que la mejora en algunos aspectos, como la gestión de errores.

La interfaz **Path** representa un archivo o directorio en Java.

No necesariamente debe corresponder con un archivo existente en el sistema de archivos. También se puede utilizar para comprobar si un archivo existe o no, o para crear un archivo nuevo, entre otros.

Tened en cuenta que la ruta de un archivo o directorio depende del sistema operativo.

Crearemos un objeto **Path** a partir de uno de los métodos **factory** de la clase **Path**:

```
Path fichero1 = Path.of("/home/usuario/ejemplo1.txt"); //Linux
Path fichero2 = Path.of("c:\\Users\\usuario\\ejemplo2.txt"); //Windows
```

En Windows, es necesario utilizar una doble barra (\\) porque el carácter \ ya tiene el significado de preservación dentro de una cadena en Java.

Podemos acceder directamente a archivos/directorios que estén en el directorio del usuario que está ejecutando nuestro programa:

```
Path archivo = Path.of(System.getProperty("user.home"), "dir",
"archivo.txt");
```

archivo apuntaría a **/home/usuario/dir/fichero.txt** en Linux/MacOS, o en **C:\Users\usuario\dir\archivo.txt** en Windows.

La interfaz **Path** nos proporciona un conjunto de métodos útiles para obtener información de la ruta que hemos especificado. Por ejemplo, **getFileName()** nos devuelve la última parte de la ruta, mientras que **getParent()** nos devuelve la ruta entera excepto la última parte.

2.3.1. LISTAR EL CONTENIDO DE UN DIRECTORIO

```
import java.io.IOException;
import java.nio.file.*;
import java.util.Scanner;

public class Ej24ListarDirectorio {
   public static void main(String[] args) {
       Scanner sc = new Scanner(System.in);
       String nombre = sc.nextLine();
       Path dir = Path.of(nombre);
       System.out.println("Ficheros del directorio " + dir);
       if (Files.isDirectory(dir)) {
           try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir);)
{
               for (Path fichero: stream) {
                   System.out.println(fichero.getFileName());
               }
           } catch (IOException | DirectoryIteratorException ex) {
               System.err.println(ex);
           }
       } else {
           System.err.println(dir.toString()+" no es un directorio");
       }
   }
}
```

2.3.2. CREAR Y BORRAR UN FICHERO

```
import java.io.IOException;
import java.nio.file.*;
import java.util.Scanner;

public class Ej232CrearBorrarFichero {
   public static void main(String[] args) {
       Scanner sc = new Scanner(System.in);
       String nombre = sc.nextLine();
       Path fichero = Path.of(nombre);
       // Files.exists(fichero)
       try {
           try {
```

```
                Files.createFile(fichero);
                System.out.println("Se ha creado el fichero");
            } catch (FileAlreadyExistsException ex) {
                System.out.println("El fichero ya existía");
            }
            System.out.println("Ruta absoluta: " +
fichero.toAbsolutePath());
            System.out.print("Quieres borrar el fichero " + fichero +
"(s/N)? ");
            if (sc.nextLine().equalsIgnoreCase("s")) {
                Files.delete(fichero);
                System.out.println("Se ha borrado el fichero.");
            } else {
                System.out.println("Se ha conservado el fichero.");
            }
        } catch (IOException ex) {
            System.out.println(ex);
        }
    }
}
```

3. FICHEROS DE TEXTO

3.1. Versión Clásica

3.1.1. LECTURA DE FICHEROS DE TEXTO

Usaremos las siguientes clases:

- File: para representar el fichero que se quiere leer.
- FileReader: Establece el stream de datos de lectura del fichero. **Lee carácter a carácter**. El constructor recibe un objeto File como parámetro.
- BufferedReader: Crea un buffer que permite leer más de un carácter. Reduce la cantidad de accesos a disco que necesitamos y se acelera notablemente la ejecución del programa. El constructor recibe un objeto FileReader como parámetro.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Ej311LectorTexto {
   public static void main(String[] args) {

           try (BufferedReader lector = new BufferedReader(new
                                                FileReader("lineas.txt"));) {
               String linea;
               while ((linea=lector.readLine()) != null)
                   System.out.println(linea);
           } catch (IOException e) {
                System.err.println(e.getMessage());
           }
   }
}
```

- **readLine()** devuelve **null** cuando hemos llegado a final de archivo.

- **readLine()** considera que una línea termina cuando encuentra cualquiera de las marcas que se utilizan para indicar final de línea: \\n, \\r o \\r\\n.

En el código anterior se está haciendo uso de la sentencia try-with-resources que es un try que declara uno o más recursos, que son objetos que deben ser cerrados después de usarlos. Esta sentencia se asegura de que sean cerrados después de la finalización del try. Es por ello que ya no se hace un close de la variable lector.

Para ello, las clases de estos objetos deben implementar la interfaz AutoCloseable, como sucede con las clases relativas a ficheros.

En el caso que no se utilizase una cláusula try-catch para proteger las operaciones sobre el fichero, el método main debería de lanzar la correspondiente excepción en su cabecera (en este caso "throws IOException").

Adicionalmente aclarar que también se podría utilizar la clase **Scanner** para leer un fichero. A diferencia de la clase FileReader que lee caracteres, la clase Scanner lee cadenas. De esta forma dispondríamos de los métodos hasNextLine() y nextLine() para leer el fichero.

```
import java.io.IOException;
import java.nio.file.Path;
import java.util.Scanner;

public class Ej311LectorTexto2 {
   public static void main(String[] args) {

        Path p = Path.of("res/palabras.txt");
        try (var lector = new Scanner(p);) {
            while (lector.hasNextLine())
                System.out.println(lector.nextLine());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

   }
}
```

Eso sí, si no se trabaja con buffer, el acceso al fichero se vería ralentizado.

3.1.2. ESCRITURA DE FICHEROS DE TEXTO

Usaremos las siguientes clases:

- File: para representar el fichero que se quiere escribir.
- FileWriter: Establece el stream de datos de escritura. El constructor recibe un objeto File como parámetro.
- BufferedWriter: Crea un buffer. Reduce la cantidad de accesos a disco que necesitamos y se acelera notablemente la ejecución del programa. El constructor recibe un objeto FileWriter como parámetro.
- PrintWriter: A diferencia de la clase FileWriter que trabaja con carácteres, la clase PrintWriter trabaja con cadenas, por lo que nos permite usar print() y println(). No utiliza buffer.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class Ej312EscritorTexto {
   public static void main(String[] args) {
       try (BufferedWriter escritor = new BufferedWriter(new
FileWriter("res/numeros.txt",true));) {
           for (int i = 1; i <=10; i++) {
               escritor.write("Línea número " + i);
               escritor.newLine();
           }
           System.out.println("Escritura realizada.");
       } catch (IOException e) {
           System.err.println(e.getMessage());
       }
   }
}
```

Hemos utilizado el método **newLine()** de **BufferedWriter** para escribir un salto de línea adecuado a la plataforma en la que se ejecuta el programa.

Hemos añadido true al FileWriter para indicar que en el caso de que el fichero ya exista, añadiremos el nuevo texto al final.

Es interesante ver que el constructor de **BufferedWriter** recibe como parámetro un FileWriter. Internamente, el **BufferedWriter** utilizará los métodos de **FileWriter** para escribir en el archivo, modificando la forma de hacerlo en su implementación. Este esquema se conoce con el nombre de Decorator Pattern.

Al abrir el fichero con **FileWriter**, podemos indicar un segundo parámetro al constructor donde indicaremos si queremos que el fichero se sobreescriba o no. Por defecto su valor es "**append=false**" y, por tanto, se sobreescribe globalmente. En el caso de que lo que se desee sea añadir contenido al fichero sin sobreescribirlo será necesario indicar en este segundo parámetro el valor true.

Adicionalmente aclarar que también se podría utilizar la clase **PrintWriter** para leer un fichero. De esta forma dispondríamos del método println que escribe y añade una línea directamente.

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class Ej312EscritorTexto2 {
   public static void main(String[] args) {
       try (PrintWriter escritor = new PrintWriter("res/numeros.txt")) {
           for (int i = 1; i <=10; i++) {
               escritor.println("Línea número  "+ i);
           }
           System.out.println("Escritura realizada.");
       } catch (FileNotFoundException e) {
           System.err.println(e.getMessage());
       }
   }
}
```

La clase PrintWriter no permite escoger si el fichero se abre exclusivamente para añadir contenido. **Con PrintWriter siempre se sobreescribirá el contenido previo**.

3.2. Versión con Java 7+

3.2.1. LECTURA DE FICHEROS DE TEXTO

En el siguiente código utilizaremos las clases Path y Files:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public static void Ej321LectorTexto(String[] args) {

   try (var file = Files.newBufferedReader(Paths.get("res/lineas.txt"))) {

       String linea;
       while ((linea = file.readLine()) != null) {
           System.out.println(linea);
       }

   } catch (IOException e) {
       System.out.println("No se ha podido abrir el fichero.");
   }
}
```

Con el método **readAllLines** de la clase Files podemos guardar todo el fichero en una colección (recomendable para ficheros pequeños!):

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

public class Ej321LectorTexto2 {

   public static void main(String[] args) {
       try {

           List<String> lineas = Files.readAllLines(Paths.get("res/lineas.txt"));
           for (String linea : lineas) {
               System.out.println(linea);
           }

       } catch (
               IOException e) {
           System.out.println("No se ha podido abrir el fichero.");
       }
   }
}
```

Con el método **readString** podemos guardar todo el fichero en un String:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class Ej321LectorTexto4 {
   public static void main(String[] args) {
       try {
           String lineas = Files.readString(Paths.get("res/lineas.txt"));
           System.out.println(lineas);
       } catch (IOException e) {
           System.out.println("No se ha podido abrir el fichero.");
       }

   }
}
```

También podemos utilizar la clase Stream existente a partir de Java 8 como destino del fichero:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.stream.Stream;

public class Ej321LectorTexto3 {
   public static void main(String[] args) {
           try (Stream<String> file = Files.lines(Paths.get("res/lineas.txt"))) {

               file.forEach(linea -> System.out.println(linea));

           } catch (IOException e) {
               System.out.println("No se ha podido abrir el fichero.");
           }

   }
}
```

3.2.2. ESCRITURA DE FICHEROS DE TEXTO

Para escribir en ficheros a partir de Java 7 podemos utilizar diferentes StandardOpenOptions:

```
public static void main(String[] args) {

  try (var file = Files.newBufferedWriter(Paths.get("numeros.txt"),
          Charset.defaultCharset(), StandardOpenOption.APPEND)) {

    for (int i = 1; i <= 10; i++) {
      file.write("BW2. Línea nº " + i);
      file.newLine();
    }
    System.out.println("Escritura realizada.");

  } catch (IOException e) {
    System.out.println("No se ha podido escribir en el fichero.");
  }
```

4. FICHEROS BINARIOS

A menudo queremos escribir en un fichero un grupo de datos en formato binario.

El formato binario tiene la ventaja de ser muy compacto: si un int ocupa 4 bytes en memoria, ocupará 4 bytes en el archivo. En cambio, ese mismo int escrito en texto podría ocupar mucho más. A cambio, un archivo binario no se puede leer fácilmente con un editor de textos y el programa que lo ha creado debe recordar qué datos y en qué orden los ha guardado para poder recuperarlos correctamente.

Para trabajar con datos Java nos proporciona las clases **DataInputStream** y **DataOutputStream**. Estas clases permiten leer y escribir cualquier tipo primitivo directamente. En este apartado, no profundizaremos en estas clases, sino en las clases **ObjectOutputStream** y **ObjectInputStream** que nos facilitan el trabajo de guardar y recuperar objetos enteros en archivos binarios.

Para que los objetos de una clase se puedan guardar y cargar utilizando este sistema es necesario que implementen la interfaz Serializable. Esta interfaz no tiene ningún método, sirve sólo para marcar que los objetos de esa clase se pueden guardar en archivos.

En este apartado trabajaremos con esta sencilla, que la única novedad que presenta es

que implementa Serializable:

```
import java.io.Serializable;

public class Mascota implements Serializable {
   private static final long serialVersionUID = 1L;

   private String nombre;
   private int numPatas;
   private boolean pelo;

   public Mascota(String nombre, int numPatas) {
       this.nombre = nombre;
       this.numPatas = numPatas;
       this.pelo = true;
   }

   public Mascota(String nombre, int numPatas, boolean pelo) {
       this(nombre,numPatas);
       this.pelo=pelo;
   }

   public String getNombre() {
       return nombre;
   }

   public int getNumPatas() {
       return numPatas;
   }

   public boolean hasPelo() {
       return pelo;
   }

   @Override
   public String toString() {
       return "Mascota nombre="+nombre+" numPatas="+ numPatas +"
pelo="+pelo;
   }

}
```

Un problema que podemos tener con la recuperación de un objeto guardado con este sistema es si la clase se ha modificado en medio. Imaginemos que guardamos una mascota como la que hemos definido en un archivo, más tarde modificamos la clase y añadimos el atributo especie y sacamos el atributo por el. Cuando intentamos volver a leer este archivo está claro que no podremos.

Para facilitar al Java la detección de estos conflictos se pone en todas las clases Serializable un atributo estático llamado **serialVersionUID** con un número de serie. Este número debe cambiarse cada vez que se hagan cambios en la clase que hagan incompatible la carga de objetos creados con versiones anteriores.

4.1. Escritura de objetos

En el siguiente ejemplo creamos un grupo de objetos Mascota y los guardamos en un archivo:

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

public class Ej41EscritorObjetos {

   public static void main(String[] args) {
       Mascota[] mascotas = new Mascota[4];
       mascotas[0] = new Mascota("Rudy", 4);
       mascotas[1] = new Mascota("Piolin", 2, false);
       mascotas[2] = new Mascota("Nemo", 0, false);
       mascotas[3] = new Mascota("Tara", 8);

       try (ObjectOutputStream escritor = new ObjectOutputStream(new
FileOutputStream("prueba.bin"));) {
           for (Mascota m : mascotas) {
               escritor.writeObject(m);
           }
       } catch (IOException ex) {
           System.err.println(ex);
       }
   }
}
```

4.2. Lectura de objetos

A continuación recuperamos las mascotas que hemos guardado en el ejemplo anterior.

Este programa supone que no conocemos cuántas mascotas hemos guardado y las va recuperando hasta que se produce una excepción de tipos EOFException, indicando que hemos intentado leer más allá del final del archivo.

```
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.ArrayList;
import java.util.List;

public class Ej42LectorObjetos {
   public static void main(String[] args) {
       List<Mascota> mascotas = new ArrayList<Mascota>();

       try (ObjectInputStream lector = new ObjectInputStream(new
FileInputStream("prueba.bin"));) {
           while (true) {
               Object o = lector.readObject();
               if (o instanceof Mascota) {
                   mascotas.add((Mascota)o);
               }
           }

       } catch (EOFException ex) {
           System.out.println("Hemos llegado al final del archivo.");
           for (Mascota m : mascotas) {
               System.out.println(m);
           }
       } catch (IOException ex) {
           System.err.println(ex);
       } catch (ClassNotFoundException ex) {
           System.err.println(ex);
       }
   }
}
```

Cuando recuperamos un objeto con **readObject** nos devuelve una referencia de tipo Object, aunque el objeto sea de otro tipo. Es necesario hacer un cast para transformar la referencia a la clase a la que realmente pertenece el objeto.

Con este sistema es posible guardar objetos de diferentes clases en un mismo archivo. A la hora de recuperarlos es útil utilizar el operador **instanceof** para tratar cada tipo de objeto de la forma adecuada.

4.3. Lectura y escritura de estructuras complejas de objetos

Una ventaja que tiene este sistema de guardar objetos en archivos es que el Java se encarga de guardar automáticamente las referencias a otros objetos y se encarga de guardar sólo una vez cada objeto si aparece más de una vez.

Naturalmente, sólo se guardarán las referencias a otros objetos de clases que también sean Serializable. Las referencias a otros objetos las deberá restaurar la aplicación en el momento de cargar los objetos. Ejemplos de objetos que no se guardan en archivo pueden ser las conexiones a la red o bases de datos, así como cualquier atributo que marquemos con el modificador **transient**.

Para resolver todos estos detalles, nuestra clase puede implementar **Externalizable** (que extiende **Serializable**) e implementar los métodos **readExternal** y **writeExternal** que se llamarán respectivamente en el proceso de leer y guardar un objeto.

Aquí no entraremos en tanto detalle y sólo comprobaremos cómo las referencias a otros objetos se guardan automáticamente.

Imaginemos la siguiente clase Persona. Una Persona puede tener varias mascotas:

```java
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Persona implements Serializable {
   private static final long serialVersionUID = 1L;

   private String nombre;
   private List<Mascota> mascotas = new ArrayList<Mascota>();

   public Persona(String nombre) {
       this.nombre=nombre;
   }

   public String getNombre() {
       return nombre;
   }

   public void addMascota(Mascota mascota) {
       mascotas.add(mascota);
   }

   public List<Mascota> getMascotas() {
       return Collections.unmodifiableList(mascotas);
   }

   @Override
   public String toString() {
       return "Persona "+nombre;
   }
}
```

El siguiente programa crea una persona que tendrá dos mascotas, guarda la persona a archivo, y cuando la recupera, podemos observar cómo también se han recuperado sus mascotas.

```
import java.io.*;

public class Ej43EstructurasComplejas {

   public static void main(String[] args) {
       Persona maria = new Persona("María");
       maria.addMascota(new Mascota("Rudy", 4));
       maria.addMascota(new Mascota("Piolin", 2, false));

       try (ObjectOutputStream escritor = new ObjectOutputStream(new
FileOutputStream("prueba.bin"));) {
           escritor.writeObject(maria);
       } catch (IOException ex) {
           System.err.println(ex);
       }

       Persona p = null;
       try (ObjectInputStream lector = new ObjectInputStream(new
FileInputStream("prueba.bin"));) {
           p = (Persona) lector.readObject();
       } catch (IOException ex) {
           System.err.println(ex);
       } catch (ClassNotFoundException ex) {
           System.err.println(ex);
       }

       if (p != null) {
           System.out.println(p);
           for (Mascota mascota : p.getMascotas()) {
               System.out.println(mascota);
           }
       }
   }
}
```

5. FICHEROS DE PARÁMETROS DE CONFIGURACIÓN

Es habitual que las aplicaciones informáticas necesiten unos parámetros de configuración (nombre de la base de datos, ubicación del servidor, idioma, etc). Java proporciona una herramienta para leer los archivos de propiedades, la clase **Properties**.

Se trata de un fichero de texto plano formado por parejas clave=valor

```
#fichero de configuración
#Wed Sep 28 21:27:07 CEST 2022
user=Manolo
password=clave37.
fichero=facturas
```

5.1. Leer ficheros de configuración

1. Inicializar el objeto Properties y se creará un fichero de propiedades.

```
Properties propiedades = new Properties();
```

2. El método **load** permite cargar en un objeto Properties los valores de las propiedades que se encuentren almacenados en un archivo.

```
propiedades.load(new FileReader("parametros.properties"));
```

3. Leer una propiedad:

```
propiedades.getProperty("user");
```

4. También podemos leerlas todas. Cargamos en un Enum todas las claves, y obtenemos su valor:

```
Enumeration<Object> claves = propiedades.keys();
while (claves.hasMoreElements()) {
Object clave = claves.nextElement();
System.out.println(clave.toString() + " - " +
propiedades.get(clave).toString());
}
```

5.2. Modificar el valor de una clave

Para modificar el valor de cualquier propiedad contenida en un objeto Properties (ojo, en el objeto, no en el archivo), debes usar el método setProperty de esa misma clase:

```
properties.setProperty("language", "FR");
```

una enumeración es una lista de constantes con nombre que definen un nuevo tipo de datos Pero el valor contenido por las propiedades de un objeto Properties se perderán si no se almacenan en un archivo. Para guardar el contenido en un archivo de tipo texto se puede usar el método store:

```
confi.store(new FileOutputStream("configuracion.properties"), "fichero de
configuracion");
confi.store(new BufferedWriter(new FileWriter ("configuracion.props")),
"Ejemplo Properties");
```

6. FICHEROS XML Y JSON CON JACKSON

6.1. JSON

6.1.1. ¿QUÉ ES JSON?

JSON es la abreviatura de JavaScript Object Notation (Notación de objetos de JavaScript). JSON es el formato de ficheros más popular para el intercambio de datos por APIs RESTful que permiten la comunicación entre aplicaciones front-end (lado cliente) y back-end (lado servidor). Adicionalmente, los navegadores interpretan nativamente el lenguaje JavaScript y pueden analizar directamente objetos JSON.

En JSON, las estructuras se presentan en las siguientes formas:

- **Objeto**. Podría decirse que es un conjunto no ordenado de pares clave/valor expresado entre { y } (llaves). La clave y el valor asociado se separan mediante : (dos puntos). Cada par se separa del siguiente mediante una coma.

```
{
  "active": true,
  "formed": 2016,
  "homeTown": "Metro City",
  "members": [
    {
      "age": 29,
      "name": "Molecule Man",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
```

```
        "Radiation blast"
      ],
      "secretIdentity": "Dan Jukes"
    },
    {
      "age": 39,
      "name": "Madame Uppercut",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ],
      "secretIdentity": "Mary Jane Wilson"
    },
    {
      "age": 1000000,
      "name": "Eternal Flame",
      "powers": [
        "Immortality",
        "Heat Immunity",
        "Inferno",
        "Teleportation",
        "Interdimensional travel"
      ],
      "secretIdentity": "Unknown"
    }
  ],
  "secretBase": "Super tower",
  "squadName": "Super hero squad"
}
```

- **Array**. Es una colección de valores encerrada por [ ] (corchetes o square brackets).

```
{
  "heroes": [
    {
      "age": 29,
      "name": "Molecule Man",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ],
```

```
      "secretIdentity": "Dan Jukes"
    },
    {
      "age": 39,
      "name": "Madame Uppercut",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ],
      "secretIdentity": "Jane Wilson"
    }
  ]
}
```

- **Valor**. Puede ser de los siguientes tipos:
  - **String**: en UTF-8. Van siempre entre dobles comillas.
  - **Number**: números. Al guardarse en BSON admite los de tipo byte, int32, in64 o double.
  - **Boolean**: true o false.
  - **Array**: entre corchetes [] y pueden contener de 1 a N elementos, que pueden ser de cualquiera de los otros tipos.
  - **Documentos**: un documento en formato JSON puede contener otros documentos embebidos que incluyan más documentos o cualquiera de los tipos anteriormente descritos.
  - **Null**.

6.1.2. VALIDEZ JSON

- Hay multitud de sitios o aplicaciones que permiten comprobar si un objeto JSON es válido o no. Un par de ejemplos:
  - JSONLint
  - JSONFormatter

6.1.3. JSON VS XML

Ejemplo employees.xml

```xml
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```

Equivalente employees.json

```json
{"employees":[

  { "firstName":"John", "lastName":"Doe" },

  { "firstName":"Anna", "lastName":"Smith" },

  { "firstName":"Peter", "lastName":"Jones" }

]}
```

Si comparamos XML con JSON podemos concluir:

- Tienen en común:
  - Tanto JSON como XML son "autodescriptivos" (legibles por humanos)
  - Tanto JSON como XML son jerárquicos/arborescentes (valores dentro de valores)
  - La mayoría de lenguajes de programación pueden analizar y utilizar tanto JSON como XML.
- Se diferencian:
  - JSON no utiliza etiquetas
  - JSON es más corto
  - JSON es más compacto y rápido de leer y escribir
  - JSON puede usar arrays o listas (por tanto, permite claves multivaluadas)

Información adicional:

- JSON es más ligero y fácil de parsear que XML. Más en detalle
- Conversor: Conversor online

6.2. JACKSON

6.2.1. ¿Qué es JACKSON?

Jackson es la librería de mapeo de objetos Java más popular que proporciona varias formas diferentes de leer y escribir ficheros JSON en lenguaje Java. Su principal ventaja es la **simplicidad** con la que realiza el mapping de un fichero JSON a un objeto Java. Para llevar a cabo este mapeo, necesitaremos disponer de una instancia **ObjectMapper**.

Jackson Tutorial | Baeldung

Repositorio código Baeldung

Three ways to use Jackson for JSON in Java (twilio.com)

Tutorial Jackson

6.2.2. ObjectMapper

ObjectMapper es la clase principal de Jackson que tiene los métodos **readValue** para leer y **writeValue** para escribir.

Aquí tenéis una breve introducción.

Y un intro sencilla de Baeldung sobre ObjectMapper.

6.2.3. Annotations

Annotations de Jackson

6.2.4. Generación automática de clases Java

Para generar POJOs de JSON: la web jsonschema2pojo o plugin de Intellij RoboPOJOGenerator (este plugin es **MARAVILLOSO e IMPRESCINDIBLE** para simplificar el mapeado de una API a clases y que el ObjectMapper no falle). En todo caso, si hay clases o atributos que no queremos que se mapean podemos utilizar la annotation **@JsonIgnoreProperties (ignoreUnknown=true)**.

Si hubiese atributos desconocidos al llamar al ObjectMapper, recordad que podemos preservar la lectura añadiendo la siguiente línea que evitará que falle la petición a la

API:

```
ObjectMapper mapper = new ObjectMapper()
      .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
MyDto readValue = mapper.readValue(jsonAsString, MyDto.class);
```

6.2.5. Dependencias Jackson

Repositorio de dependencias Maven

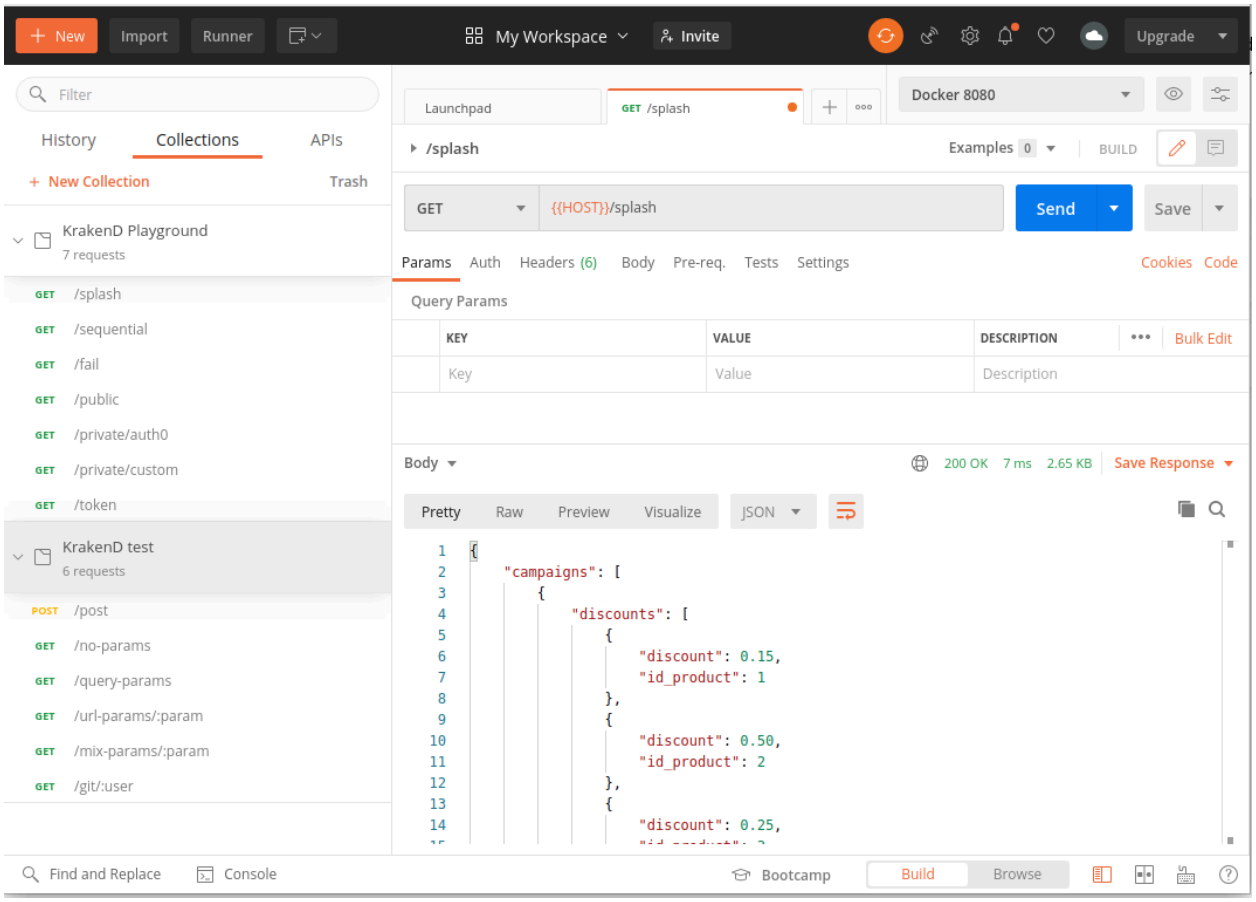
Dependencias:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.14.0-rc1</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.14.0-rc1</version>
</dependency>
```

6.2.6. Consulta de APIs con PostMan

PostMan es una herramienta súper útil para gestionar el consumo y las peticiones a APIs: Postman (instalar versión desktop).

Dentro de colecciones podremos agrupar las diferentes request que hagamos a APIs de nuestro interés y analizar sus respuestas.



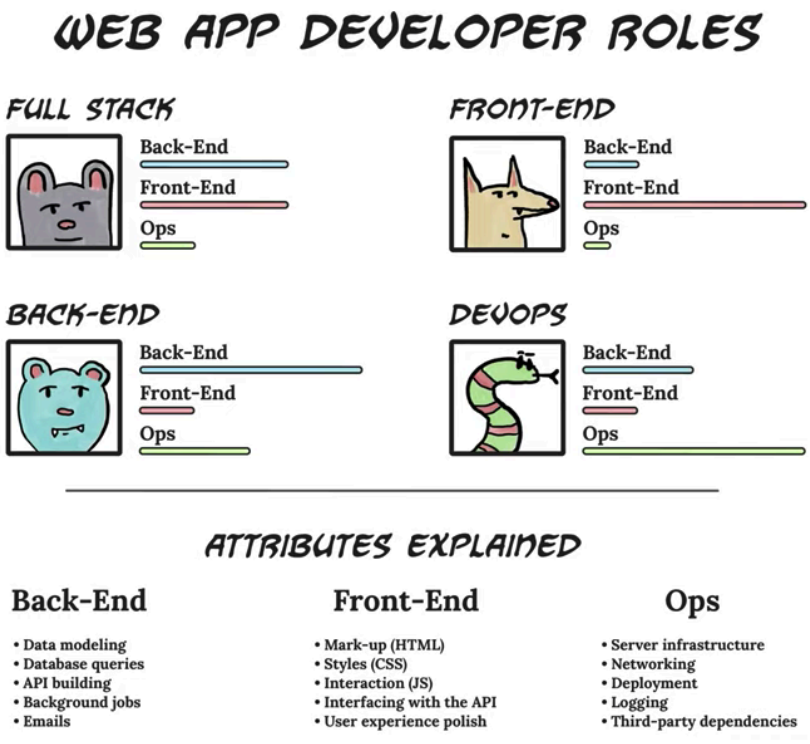
Listado de [Public APIs]

[Cool APIs]

[¿Qué es una API RESTful?]

EXTRA: Web Developer Jobs

[Wikipedia Front-End vs Back-End]



[Developers Survey 2022 de StackOverFlow]

EXTRA1. FICHEROS BYTE A BYTE

Habitualmente nunca trabajaremos con un archivo byte a byte, pero los demás accesos a archivos que haremos trabajan internamente utilizando las clases que veremos en este apartado.

Todos los flujos de datos (streams) trabajan a partir de las clases InputStream y OutputStream. Estas dos clases son **abstractas** y las API nos proporcionan un grupo de clases derivadas especializadas en la lectura y escritura de varios tipos de flujos de datos.

Para la lectura y escritura de archivos de bytes se utilizan las clases **FileInputStream** y **FileOutputStream**.

E1.1. Escritura de un archivo byte a byte

En este ejemplo escribimos un grupo de bytes en un archivo:

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

public class EjExtra11EscritorBytes {

   public static void main(String[] args) {
       /* Creando el objeto en el try hacemos que se llame a su método
close() automáticamente cuando el try termina, aunque salte una excepción.
Esto se puede hacer con cualquier clase que implemente Closeable. Si no
hacemos así, deberíamos añadir una cláusula finally después de los catch y
hacer escritor.close() allí.*/

     try(FileOutputStream escritor = new
FileOutputStream("res/prueba.txt");) {
       for (char ch : "hola".toCharArray()) {
           escritor.write(ch);
       }
       for (byte b=65; b<80; b++) {
           escritor.write(b);
       }
       byte[] bytes = {97, 98, 99, 100, 101, 102, 103, 104, 105};
       escritor.write(bytes, 2, 5); // escribe 5 bytes empezando por la
posición 2.
```

```
    } catch (FileNotFoundException e) {
            // No quiere decir que no exista, sino que no puede escribir.
            System.err.println(e);
    } catch (IOException e) {
        // Podríamos haber puesto sólo esta excepción, porque
 FileNotFounException
        // deriva de IOException.
        System.err.println(e);
    }
    }
 }
```

Si abrimos el archivo prueba.txt después de la ejecución de este programa encontraremos el texto "holaABCDEFGHIJKLMNOcdefg".

Esto funciona porque estos caracteres se codifican en UTF-8 utilizando uno solo byte, que coincide con el código ASCII, pero no funciona con otros caracteres y símbolos, que ocupen más de un byte (un char en Java ocupa 2 bytes).

Es decir, **ésta no es la forma correcta de escribir texto en un archivo**.

E1.2. Lectura de un archivo byte a byte

En el siguiente ejemplo leemos el archivo que hemos creado en el apartado anterior.

Lo leeremos dos veces: en la primera lo leeremos byte a byte, mientras que en la segunda lo leeremos entero y guardaremos los bytes leídos en un array.

```
import java.io.FileNotFoundException;
import java.io.FileInputStream;
import java.io.IOException;

public class EjExtra12LectorBytes {

    public static void main(String[] args) {
        int b=0;

        try (FileInputStream lector = new FileInputStream("prueba.txt");) {
            // Lee byte a byte. -1 indica final de archivo.
            while (b!=-1) {
                b = lector.read();
```

```
                if (b!=-1)
                    System.out.print(b+" ");
            }
            System.out.println();
        } catch (FileNotFoundException e) {
            System.err.println(e);
        } catch (IOException e) {
            System.err.println(e);
        }

        byte[] contenido = new byte[100];
        int leídos;
        try (FileInputStream lector = new FileInputStream("prueba.txt");) {
            // Lee todo el archivo y lo guarda en un array.
            leídos = lector.read(contenido);
            if (leídos==-1) {
                System.out.println("Se ha llegado al final del archivo.");
            } else {
                System.out.println("Se han leído "+leídos+" bytes.");
            }
        } catch (FileNotFoundException e) {
            System.err.println(e);
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

EXTRA2. COMPARATIVA DE VELOCIDAD ENTRE FILEWRITER Y BUFFEREDWRITER

En este programa comparamos la velocidad de ejecución entre un **FileWriter** y un **BufferedWriter**.

Aprovechando que ambos derivan de **Writer** hemos creado un método que recibe un Writer cualquiera y lo utiliza para escribir una línea 10 millones de golpes.

Con **System.currentTimeMillis()** obtenemos el tiempo en el momento en que el llamamos. Mirando el tiempo antes de empezar y justo cuando terminamos el proceso, y restando los dos tiempos, podemos saber cuándo ha tardado en realizar la operación.

```
import java.io.BufferedWriter;
```

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;

public class TestVelocidadEscritura {

	public static void main(String[] args) {
	    try (BufferedWriter bufferedWriter =
	                new BufferedWriter(new
FileWriter("pruebaBuffered.txt"));) {
	        long tiempo = testVelocidad(bufferedWriter);
	        System.out.println("pruebaBuffered.txt creado en " + tiempo
+ " ms");
	    } catch (IOException e) {
	        System.err.println(e);
	    }

	    try (FileWriter fileWriter = new FileWriter("pruebaFile.txt");)
{
	        long tiempo = testVelocidad(fileWriter);
	        System.out.println("probaFile.txt creado en " + tiempo + "
ms");
	    } catch (IOException e) {
	        System.err.println(e);
	    }
	}

	public static long testVelocidad(Writer writer) throws IOException {
	    long tiempoInicial=System.currentTimeMillis();
	    for(int i = 0; i < 10000000; i++) {
	        writer.write("prueba");
	        writer.write('\n');
	    }
	    long tiempoFinal=System.currentTimeMillis();
	    return tiempoFinal-tiempoInicial;
	}
   }
```

Por tanto, queda demostrado que un BufferedWriter es 1,5 más rápido que un

FileWriter.