DOCUMENTACIÓN DEL TRABAJO DE ACCESO A DATOS REALIZADO POR IVÁN AZAGRA Y DANIEL GARCÍA

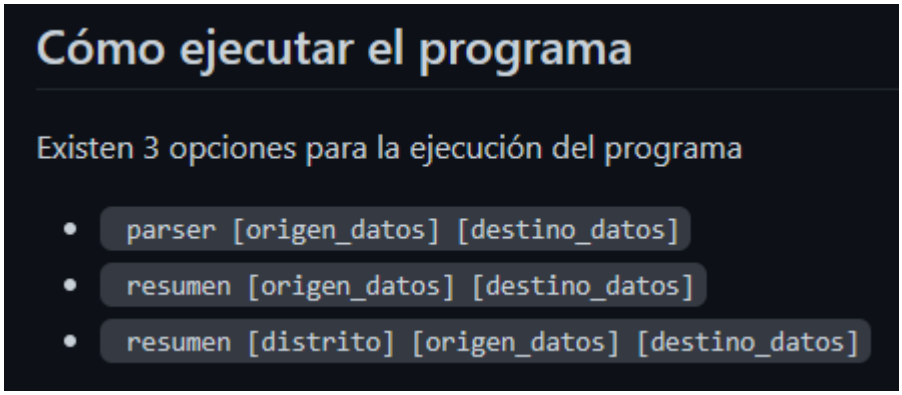
Trabajo acceso a datos

Iván Azagra y Daniel Rodríguez

Contenido

Cómo ejecutar el programa:

A la hora de ejecutar la aplicación esta debe ser llamada por consola añadiendo los argumentos necesarios para la consulta, estos argumentos son la carpeta de origen de los datos y la carpeta donde se guardarán los archivos generados durante la ejecución, así como en el caso de querer consultar un distrito específico añadir este valor, aunque este último es opcional. Se debe respetar el orden puesto a continuación:



En caso de no introducir los datos correctamente se finalizará la ejecución.
En la carpeta de origen de datos deberán encontrarse los csv que se utilizan para sacar los datos, estos son contenedores_varios.csv y modelo_residuos_2021.csv.

¡Se necesitan ambos archivos para el correcto funcionamiento del programa!

El programa guardará un archivo de la ejecución del programa en caso exitoso o fallido, en él se podrá consultar el tiempo de ejecución, fecha de uso y tipo de opción ejecutada.

La opción de parseado solo admite las carpetas de origen y destino en ese orden, esta opción coge los archivos con formato csv que se encuentren en el directorio de origen y los formatea y organiza a arhivos XML y JSON en la carpeta que hayamos pasado como destino.

```
class CSVParser(private val originalDirectory: String, private val destinationDirectoryPath: String) {
    private val destinationDirectory = File(destinationDirectoryPath)
    private val directory = File(originalDirectory)
    //private val dtf: DateTimeFormatter = DateTimeFormatter.ofPattern("dd_mm_yyyy")
    //private val now: LocalDateTime = LocalDateTime.now()

    fun parse(): Int {
        if (originalDirectory != "${System.getProperty("user.dir")}${File.separator}data") {
            val fileResid = File("$originalDirectory${File.separator}modelo_residuos_2021.csv")
            val fileContened = File("$originalDirectory${File.separator}contenedores_varios.csv")
            val files = directory.listFiles()
            if (files == null) {
                println("directory $directory is empty.")
                exitProcess(99_999)
            }
            if (files.contains(fileResid) && files.contains(fileContened)) { } else {
                println("Directory $directory is not the directory which contains the CSV files.")
                exitProcess(7777)
            }
        }

        createDirectory()

        if (!directory.exists()) {
            println("$originalDirectory does not exist.")
            exitProcess(1710)
        } else if (!directory.isDirectory) {
            println("$originalDirectory is not a directory.")
            exitProcess(1709)
        }
        if (directory.listFiles() == null) {
            println("directory $directory is empty.")
            exitProcess(99_999)
        }
```

```
private fun parseCSVContenedores(file: File): Int {
    val contenedores = CSVReader.readCSVContenedores(file.absolutePath, ";")
    val newFile = File("${destinationDirectoryPath}${File.separator}contenedores_varios_parsed.csv")
    val writer = FileWriter(newFile)
    writer.write("Código Interno del Situad;Tipo Contenedor;Modelo;Descripcion Modelo;Cantidad;Lote;Distrito;Barrio;Tipo Vía;Nombre;Número\n")
    contenedores.forEach {
        val line = "${it.codigoSituado};${it.tipoContenedor};${it.modelo};${it.descripcionModelo};${it.cantidad};" +
                "${it.lote};${it.distrito};${it.barrio};${it.tipoVia};${it.nombreCalle};${it.numero}\n"
        writer.write(line)
    }
    if (!newFile.canWrite()) {
        return 1
    }
    return 0
}

private fun parseJSONContenedores(file: File): Int {
    val contenedoresListDTO = ContenedoresListDTO(CSVReader.readCSVContenedores(file.absolutePath, ";"))
    val jsonText = GsonBuilder().setPrettyPrinting().create().toJson(contenedoresListDTO)
    val newFile = File("${destinationDirectoryPath}${File.separator}contenedores_varios_parsed.json")
    val writer = FileWriter(newFile)
    writer.write(jsonText)
    if (!newFile.canWrite()) {
        return 1
    }
    return 0
}

@Throws(JAXBException::class)
private fun parseXMLContenedores(file: File): Int {
    val contenedoresListDTO = ContenedoresListDTO(CSVReader.readCSVContenedores(file.absolutePath, ";"))
    val jaxbContext: JAXBContext = JAXBContext.newInstance(ContenedoresListDTO::class.java)
    val marshaller = jaxbContext.createMarshaller()
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true)
    val newFile = File("${destinationDirectoryPath}${File.separator}contenedores_varios_parsed.xml")
    marshaller.marshal(contenedoresListDTO, newFile)
    if (!newFile.canWrite()) {
        return 1
    }
    return 0
}

private fun parseJSONResiduos(file: File): Int {
    val residuosListDTO = ResiduosListDTO(CSVReader.readCSVResiduos(file.absolutePath, ";"))
    val jsonText = GsonBuilder().setPrettyPrinting().create().toJson(residuosListDTO)
    val newFile = File("${destinationDirectoryPath}${File.separator}modelo_residuos_2021_parsed.json")
    val writer = FileWriter(newFile)
    writer.write(jsonText)
    if (!newFile.canWrite()) {
        return 1
    }
    return 0
}

@Throws(JAXBException::class)
private fun parseXMLResiduos(file: File): Int {
    val residuosListDTO = ResiduosListDTO(CSVReader.readCSVResiduos(file.absolutePath, ";"))
    val jaxbContext: JAXBContext = JAXBContext.newInstance(ResiduosListDTO::class.java)
    val marshaller = jaxbContext.createMarshaller()
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true)
    val newFile = File("${destinationDirectoryPath}${File.separator}modelo_residuos_2021_parsed.xml")
```

Para la opción de resumen el programa admite dos opciones, una con solo los directorios de origen y destino en ese orden y otra añadiendo el distrito que se quiera consultar, esta opción llama al procesador de datos que mediante dataframes y lets-plot obtiene los datos y crea gráficas. Esta clase hace uso del lector de csv para obtener los datos en una lista que luego utilizará en los dataframes para hacer las consultas propuestas.

DataProcessor:

Clase que se dedica a procesar los datos para sacar las consultas y gráficos, cuenta de tres métodos.

- El primer método es dataToDataFrame() en el que se hacen la mayoría de las consultas requeridas.

```
fun dataToDataFrame() {
    if(!imagesPath.exists())
        imagesPath.mkdir()

    contDataframe.schema().print()
    residuoDataframe.schema().print()

    // Tipo de contenedor por distrito
    contenedoresDistrito = contDataframe.groupBy("tipoContenedor", "distrito").count().html()

    // Jose Luis te he fallado no sé cómo hacer la media de contenedores por distrito

    mediaToneladasAnuales = residuoDataframe.groupBy("year", "nombreDistrito", "tipoResiduo")
        .aggregate {
            mean("toneladas").roundToInt() into "Media de toneladas anuales por distrito"
        }.html()

    MaxMinMeanStd = residuoDataframe.groupBy("nombreDistrito","toneladas", "tipoResiduo", "year")
        .aggregate {
            max("toneladas") into "Max toneladas"
            min("toneladas") into "Min toneladas"
            mean("toneladas").roundToInt() into "Media toneladas"
            std("toneladas") into "Desviacion toneladas"
        }.sortBy("nombreDistrito").html()

    sumaRecogidoDistrito = residuoDataframe.groupBy("nombreDistrito","toneladas", "year")
        .aggregate {
            sum("toneladas") into "Suma de toneladas recogidas"
        }.html()

    cantidadTipoRecogido = residuoDataframe.groupBy("nombreDistrito","tipoResiduo")
        .aggregate{
            sum("toneladas") into "Cantidad de toneladas"
        }.html()
}
```

- El segundo método es graphics() donde se generan todos los gráficos requeridos por resumen.

```
fun graphics() {
    var d = contDataframe.groupBy("distrito", "tipoContenedor").count().toMap()
    val gTotalContenedores: Plot = letsPlot(data = d) + geomBar(
        stat = identity,
        alpha = 0.4,
        fill = Color.BLUE
    ) {
        x = "distrito"
        y = "count"
    } + labs(
        x = "distrito",
        y = "contendores por distrito",
        title = "Número de contenedores por distrito"
    )
    ggsave(gTotalContenedores, "Contenedores-Por-Distrito.png", 1, 2, imagesPath.toString())

    var d2 = residuoDataframe.groupBy("mes", "distrito").aggregate {
        mean("toneladas") into "media de toneladas mensuales"
    }.toMap()
    val gMediaToneladasMensuales: Plot = letsPlot(data = d2) +geomBar(
        stat = identity,
        alpha = 0.8,
        fill = Color.CYAN,
    ) {
        x = "mes"
        y = "media de toneladas mensuales"
    } + labs(
        x = "mes",
        y = "toneladas",
        title = "Media de toneladas mensuales por distrito"
    )
    ggsave(gMediaToneladasMensuales, "Media-Toneladas-Mensuales-Distrito.png", 1, 2, imagesPath.toString())

}
```

- El tercer método es graphicsDistrito() donde se generan los gráficos requeridos para el resumen distrito.

```
fun graphicsDistrito() {
    val d = residuoDataframe.groupBy("mes")
        .aggregate {
            max("toneladas") into "Max toneladas"
            min("toneladas") into "Min toneladas"
            mean("toneladas") into "Media toneladas"
            std("toneladas") into "Desviacion toneladas" // Puede salir NaN por algún motivo
        }.toMap()
    val gMaxMinMeanStd = letsPlot(data = d) + geomBar(
        stat = identity,
        alpha = 1,
        fill = Color.GREEN
    ) {
        x = "mes"
        y = "Max toneladas"
    }+ geomBar(
        stat = identity,
        alpha = 0.8,
        fill = Color.RED
    ) {
        x = "mes"
        y = "Min toneladas"
    } +geomBar(
        stat = identity,
        alpha = 0.9,
        fill = Color.BLUE
    ) {
        x = "mes"
        y = "Media toneladas"
    } + labs(
        x = "mes",
        y = "toneladas",
        title = "Maximo, minimo y media de residuos"
    )

    ggsave(gMaxMinMeanStd, "Max-Min-Media-Desviacion-Distrito.png", path = imagesPath.toString())

    var d2 = residuoDataframe.groupBy("nombreDistrito", "toneladas", "tipoResiduo")
        .aggregate {
            sum("toneladas") into "Total de toneladas por residuo"
        }.toMap()

    val gTotalToneladas:Plot = letsPlot(data = d2) + geomBar(
        stat = identity,
        alpha = 0.3,
        fill = Color.CYAN
    ) {
        x = "tipoResiduo"
        y = "Total de toneladas por residuo"
    }  +geomBar(
        stat = identity,
        alpha = 0.3,
        fill = Color.DARK_BLUE
    ) {
        x = "tipoResiduo"
        y = "toneladas"
    }+ labs(
        x = "tipo de residuo",
        y = "toneladas",
        title = "Toneladas totales por tipo de residuo"
    )
    ggsave(gTotalToneladas, "Toneladas-Totales-Tipo-Residuo.png", 1, 1, imagesPath.toString())
}
```

Hemos optado por la utilización de DataFrames y letsplot para la realización de esta clase, ya que era la forma más rápida tanto a la hora de programarlo como en la ejecución, ya que usando api stream teníamos una clase larguísima e innecesariamente complicada para sacar los resultados.

TemplateGenerator:

Clase que se dedica a generar un template html en el que se depositan todos los datos de las consultas requeridos para la generación del documento que se mostrará en el navegador, consta de dos métodos.

- El primer método es generateHTML() en el que se genera el archivo html en el que después se escribirá en el siguiente método
- El segundo método es generateSummary() en el que escribe el contenido

Ambos métodos son funciones generateSummary tienen un HTML creado dentro que devuelven y son usados en la función generateHTML para generar el archivo que después será ejecutado en el navegador para mostar los datos y gráficos en ella.

CSVReader:

Clase que se encarga de leer los csv y chequear si es posible leerlos, esta clase introduce los datos de los csv leídos a una lista que después usamos para el procesado de datos en DataProcessor. La clase cuenta también con los parseadores de tipos de residuo y contenedor para evitar los conflictos que dan los enumeradores al no ser Strings comparables con los valores del csv, también por esto mismo usamos los enums con un String acoplado a ellos para poder hacer esa comparación entre valor y tipo y poder hacer la distinción posterior para pasarlo a la lista.
También cuenta con una función que evita el doble delimitador para ahorrarnos los valores nulos.

```kotlin
 * @author Daniel Rodriguez
 * Este metodo coge el CSV de residuos y devuelve una lista no mutable de objetos Residuos
 * @param csvName nombre del archivo csv
 * @param delimiter tipo de delimitador que usa el csv
 */
fun readCSVResiduos(csvName: String, delimiter: String) : List<Residuos> {
    val results = mutableListOf<Residuos>()

    checkCSVResiduosIsValid(csvName)

    val lines = File(csvName).readLines().drop(1)

    if (lines.isEmpty()) {
        throw Exception("File $csvName's content is empty. Use a valid csv file.")
    }

    lines.forEach {
        val arguments = noDoubleDelimiter(it,delimiter).split(delimiter)
        val residuo = Residuos(
            year = arguments[0],
            mes = arguments[1],
            lote = arguments[2].toInt(),
            tipoResiduo = parseTipoResiduo(arguments[3]),
            distrito = arguments[4].toInt(),
            nombreDistrito = arguments[5],
            toneladas = arguments[6].replace(",", ".").toDouble()
        )
        results.add(residuo)
    }

    return results
}
```

```kotlin
/**
 * @author Iván Azagra Troya
 * método dedicado a leer el csv de contenedores
 */
fun readCSVContenedores(csvName: String, delimiter: String): List<Contenedor> {
    val results = mutableListOf<Contenedor>()

    checkCSVContenedoresIsValid(csvName)

    val lines = File(csvName).readLines().drop(1)

    if (lines.isEmpty()){
        throw Exception("File $csvName's content is empty. Use a valid csv file.")
    }

    lines.forEach {
        val arguments = noDoubleDelimiter(it,delimiter).split(delimiter)
        val contenedor = Contenedor(
            arguments[0],
            parseTipoContenedor(arguments[1]),
            arguments[2],
            arguments[3],
            arguments[4].toInt(),
            arguments[5].toInt(),
            arguments[6],
            arguments[7],
            arguments[8],
            arguments[9],
            arguments[10]
        )
        results.add(contenedor)
    }

    return results
}
```

BitacoraCreator:

En esta clase nos encargamos de crear el archivo xml donde se guardan los datos de ejecución, consta de 3 funciones.

- saveIntoBitacora(): Coge los datos del POKO de ejecución y los introduce en el archivo, comprueba si existe el directorio y si no es así lo crea, también si no existe el archivo llama al método createCosas() para que se encargue de introducir los datos usando una lista de ejecuciones. Si existe el archivo pasa los datos con el método resultsToXML()
- resultsToXML(): se encarga de pasar los datos usando las funciones de la librería JAXB
- createCosas(): Coge las ejecuciones de una lista y los introduce en el XML

```kotlin
object BitacoraCreator {
    private val PATH_TO_BITACORA_XML = "${System.getProperty("user.dir")}${File.separator}bitacora${File.separator}bitacora.xml"

    @Throws(JAXBException::class)
    fun saveIntoBitacora(execution: Ejecucion) {
        val bitacoraDirectoryURI = "${System.getProperty("user.dir")}${File.separator}bitacora"
        val bitacoraDirectory = File(bitacoraDirectoryURI)
        if (!bitacoraDirectory.exists()) {
            bitacoraDirectory.mkdirs()
        }
        val file = File(PATH_TO_BITACORA_XML)
        if (!file.exists()) {
            createCosas(execution)
        } else {
            val jaxbContext: JAXBContext = JAXBContext.newInstance(Ejecuciones::class.java)
            val unmarshaller = jaxbContext.createUnmarshaller()
            val executions: Ejecuciones = unmarshaller.unmarshal(File(PATH_TO_BITACORA_XML)) as Ejecuciones
            var duplicate = false
            for (res in executions.resultList) {
                if (res.id == execution.id) {
                    duplicate = true
                }
            }
            if (!duplicate) {
                executions.resultList.add(execution)
            }
            resultsToXML(executions)
        }
    }

    @Throws(JAXBException::class)
    private fun resultsToXML(executions: Ejecuciones) {
        val jaxbContext: JAXBContext = JAXBContext.newInstance(Ejecuciones::class.java)
        val marshaller = jaxbContext.createMarshaller()
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE)
        marshaller.marshal(executions, File(PATH_TO_BITACORA_XML))
    }

    @Throws(JAXBException::class)
    private fun createCosas(execution: Ejecucion) {
        val ejecuciones = Ejecuciones()
        ejecuciones.resultList.add(execution)
        resultsToXML(ejecuciones)
    }
}
```

Contenedor:

Clase POKO de contenedor, implementa las anotaciones para poder funcionar con JAXB. Esta clase guarda los datos de los contenedores que se encuentran en el csv y que nos eran necesarios, por lo que hemos prescindido de los valores no necesarios como las coordenadas y la dirección, ya que en caso de necesitar esta última se puede sacar simplemente concatenando los strings que componen este mismo campo. En la clase igualamos los campos con un constructor()

```
@XmlRootElement(name = "contenedor_data")
@XmlAccessorType(XmlAccessType.FIELD)
@DataSchema
class Contenedor() {
    constructor(
        codigoSituado: String = "",
        tipoContenedor: TipoContenedor,
        modelo: String = "",
        descripcionModelo: String = "",
        cantidad: Int,
        lote: Int,
        distrito: String = "",
        barrio: String = "",
        tipoVia: String = "",
        nombreCalle: String = "",
        numero: String = ""
    ) : this() {
        this.codigoSituado = codigoSituado
        this.tipoContenedor = tipoContenedor
        this.modelo = modelo
        this.descripcionModelo = descripcionModelo
        this.cantidad = cantidad
        this.lote = lote
        this.distrito = distrito
        this.barrio = barrio
        this.tipoVia = tipoVia
        this.nombreCalle = nombreCalle
        this.numero = numero
    }

    var codigoSituado: String = ""
    var tipoContenedor: TipoContenedor = TipoContenedor.RESTO
    var modelo: String = ""
    var descripcionModelo: String = ""
    var cantidad: Int = 0
    var lote: Int = 0
    var distrito: String = ""
    var barrio: String = ""
    var tipoVia: String = ""
    var nombreCalle: String = ""
    var numero: String = ""
}
```

Ejecucion:

Clase POKO de ejecución con las anotaciones de JAXB, contiene el constructor donde se le pasan los parámetros de la clase y las anotaciones en los atributos necesarios para pasárselo al XML.

```
 * @author Daniel Rodriguez Muñoz
 * Clase POKO para el objeto de la ejecución
 */
@XmlRootElement(name = "execution_data")
@XmlAccessorType(XmlAccessType.FIELD)
class Ejecucion() {
    constructor(selectedOption: String, inicioEjecucion: Long, successfullExecution: Boolean) : this()
    {
        this.selectedOption = selectedOption
        this.executionTime = System.currentTimeMillis() - inicioEjecucion
        this.wasSuccessful = successfullExecution
    }

    @XmlAttribute(name = "id")
    val id: UUID = UUID.randomUUID()
    @XmlAttribute(name = "execution_time")
    var executionTime = 0L
    val instant: String = Util.getCurrentInstantForExecution()
    var selectedOption: String = ""
    var wasSuccessful = false
}
```

Ejecuciones:

Clase POKO que consta únicamente de una lista de objetos Ejecucion. Hace la función de elemento “root” del XML que será generado por JAXB.

```
@XmlRootElement(name="ejecuciones")
@XmlAccessorType(XmlAccessType.FIELD)
class Ejecuciones() {
    @XmlElementWrapper(name = "result_list")
    val resultList : MutableList<Ejecucion> = mutableListOf()
}
```

Residuos:

Clase POKO con los datos de los residuos del csv necesarios para las consultas, contiene las anotaciones DataSchema para usar DataFrames, y las anotaciones para usarlo con JAXB y pasar esos atributos al XML que generamos en ejecución.

```
@XmlRootElement(name = "residuos_data")
@XmlAccessorType(XmlAccessType.FIELD)
@DataSchema
class Residuos() {
    constructor(
        lote: Int,
        year: String,
        mes: String,
        tipoResiduo: TipoResiduo,
        distrito: Int,
        nombreDistrito: String,
        toneladas: Double
    ): this() {
        this.lote = lote
        this.year = year
        this.mes = mes
        this.tipoResiduo = tipoResiduo
        this.distrito = distrito
        this.nombreDistrito = nombreDistrito
        this.toneladas = toneladas
    }

    @XmlAttribute(name = "lote")
    var lote: Int = 0
    @XmlAttribute(name = "year")
    var year: String = ""
    @XmlAttribute(name = "month")
    var mes: String = ""
    var tipoResiduo: TipoResiduo = TipoResiduo.RESTO
    var distrito: Int = 0
    var nombreDistrito: String = ""
    var toneladas: Double = 0.0
}
```

Tipos:

Los pongo juntos porque su contenido es muy pequeño, básicamente se limitan a ser un enumerador para comparar los tipos de contenedor y residuos que parseamos en el lector de csv.

```
enum class TipoContenedor(tipo: String) {
    ORGANICA("ORGANICA"),
    RESTO("RESTO"),
    ENVASES("ENVASES"),
    VIDRIO("VIDRIO"),
    PAPEL_Y_CARTON("PAPEL_Y_CARTON")

}
```

```
enum class TipoResiduo(tipo: String) {
    RESTO("Restos"),
    ENVASES("Envases"),
    VIDRIO("Vidrio"),
    ORGANICA("Orgánica"),
    PAPEL_Y_CARTON("Papel y cartón"),
    PUNTOS_LIMPIOS("Punto limpio"),
    CARTON_COMERCIAL("Cartón comercial"),
    VIDRIO_COMERCIAL("Vidrio comercial"),
    PILAS("Pilas"),
    ANIMALES_MUERTOS("Animales muertos"),
    RCD("RCD"),
    CONTENEDORES_DE_ROPA_USADA("Contenedores de ropa usada"),
    RESIDUOS_DEPOSITADOS_EN_MIGAS_CALIENTES("Residuos depositados en migas calientes"),

}
```

CSVParser:

Clase encargada de parsear los csv (compatibles con el modelo de residuos o contenedores) de la carpeta origen, pasándose a csv sin valores nulos ni dobles delimitadores, a xml y a json, en la carpeta de destino que se le haya asignado por constructor.

Antes de empezar a leer, se asegura de que el directorio de origen sea un directorio, exista y no esté vacío y que contenga ambos csv. De lo contrario sale del programa.

Para el directorio destino, si existe, se asegura de que sea un directorio, y si lo es, pide por consola confirmación para borrarlo, ya que, si existe previamente, lo borrará para dar paso a la nueva ejecución. Si el usuario se lo deniega, sale del programa, y si al intentar borrarlo no pudiera, sale del programa con el código 1708. Tras un borrado exitoso, este método se llama a sí mismo recursivamente. Finalmente, cuando el directorio no exista, crea dicho directorio.

Una vez está todo esto listo, procede a hacer su función principal, parsear:

Esta clase utiliza corrutinas debido a que parsear a los tres tipos de archivo es una tarea que puede ser realizada concurrentemente. De esta manera, se usa un runBlocking{ } para bloquear la ejecución de las corrutinas y que dejen de existir después de que termine dicho runBlocking. Acto seguido, se lanzan tres corrutinas mediante launch { } en el Dispatcher de Input/Output (porque como van a hacer operaciones de lectura/escritura, es el que más sentido tiene usar), y cada corrutina parsea a un tipo de archivo distinto: la que parsea a xml usa JAXB, la que parsea a json usa Gson y la que parsea a csv lee el csv pasándolo a lista de objetos y esa misma lista la vuelca en el csv destino. Una vez acaban las tres, se unen mediante join y terminan. Si todo salio bien, el metodo parse() devolverá un 0; y por cada cosa que haya salido mal, devolverá un número mayor.

```
private fun parseCSVResiduos(file: File): Int {
    val residuos = CSVReader.readCSVResiduos(file.absolutePath, ";")
    val newFile = File("${destinationDirectoryPath}${File.separator}modelo_residuos_2021_parsed.csv")
    val writer = FileWriter(newFile)
    writer.write("Año;Mes;Lote;Residuo;Distrito;Nombre Distrito;Toneladas\n")
    residuos.forEach {
        val line = "${it.year};${it.mes};${it.lote};${it.tipoResiduo};${it.distrito};${it.nombreDistrito};${it.toneladas}\n"
        writer.write(line)
    }
    if (!newFile.canWrite()) {
        return 1
    }
    return 0
}

private fun parseCSVContenedores(file: File): Int {
    val contenedores = CSVReader.readCSVContenedores(file.absolutePath, ";")
    val newFile = File("${destinationDirectoryPath}${File.separator}contenedores_varios_parsed.csv")
    val writer = FileWriter(newFile)
    writer.write("Código Interno del Situad;Tipo Contenedor;Modelo;Descripcion Modelo;Cantidad;Lote;Distrito;Barrio;Tipo Vía;Nombre;Número\n")
    contenedores.forEach {
        val line = "${it.codigoSituado};${it.tipoContenedor};${it.modelo};${it.descripcionModelo};${it.cantidad};" +
                "${it.lote};${it.distrito};${it.barrio};${it.tipoVia};${it.nombreCalle};${it.numero}\n"
        writer.write(line)
    }
    if (!newFile.canWrite()) {
        return 1
    }
    return 0
}

private fun parseJSONContenedores(file: File): Int {
    val contenedoresListDTO = ContenedoresListDTO(CSVReader.readCSVContenedores(file.absolutePath, ";"))
    val jsonText = GsonBuilder().setPrettyPrinting().create().toJson(contenedoresListDTO)
    val newFile = File("${destinationDirectoryPath}${File.separator}contenedores_varios_parsed.json")
    val writer = FileWriter(newFile)
    writer.write(jsonText)
    if (!newFile.canWrite()) {
        return 1
    }
    return 0
}

@Throws(JAXBException::class)
private fun parseXMLContenedores(file: File): Int {
    val contenedoresListDTO = ContenedoresListDTO(CSVReader.readCSVContenedores(file.absolutePath, ";"))
    val jaxbContext: JAXBContext = JAXBContext.newInstance(ContenedoresListDTO::class.java)
    val marshaller = jaxbContext.createMarshaller()
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true)
    val newFile = File("${destinationDirectoryPath}${File.separator}contenedores_varios_parsed.xml")
    marshaller.marshal(contenedoresListDTO, newFile)
    if (!newFile.canWrite()) {
        return 1
    }
    return 0
}

private fun parseJSONResiduos(file: File): Int {
    val residuosListDTO = ResiduosListDTO(CSVReader.readCSVResiduos(file.absolutePath, ";"))
    val jsonText = GsonBuilder().setPrettyPrinting().create().toJson(residuosListDTO)
    val newFile = File("${destinationDirectoryPath}${File.separator}modelo_residuos_2021_parsed.json")
    val writer = FileWriter(newFile)
    writer.write(jsonText)
    if (!newFile.canWrite()) {
        return 1
    }
    return 0
}

@Throws(JAXBException::class)
private fun parseXMLResiduos(file: File): Int {
    val residuosListDTO = ResiduosListDTO(CSVReader.readCSVResiduos(file.absolutePath, ";"))
    val jaxbContext: JAXBContext = JAXBContext.newInstance(ResiduosListDTO::class.java)
    val marshaller = jaxbContext.createMarshaller()
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true)
```

Util:

Clase Object que se usa para obtener los tiempos de ejecución y la fecha del momento dado en formato español.

```
object Util {
    fun getCurrentInstantForExecution(): String {
        val instanteEjecucion: LocalDateTime = LocalDateTime.now()
        val formatter: DateTimeFormatter = DateTimeFormatter.ISO_LOCAL_DATE_TIME
        return formatter.format(instanteEjecucion)
    }

    fun getCurrentDateTimeSpanishFormat(): String {
        val instanteEjecucion: LocalDateTime = LocalDateTime.now()
        val formatter: DateTimeFormatter = DateTimeFormatter
            .ofLocalizedDateTime(FormatStyle.MEDIUM)
            .withLocale(Locale("es", "ES"))
        return formatter.format(instanteEjecucion)
    }
}
```

Main:

La clase main se encarga de la ejecución de todas las clases y sus funciones, se crea un atributo de clase Ejecucion en la que se registra la ejecución actual, con su tiempo y opción elegida. Compara el número de argumentos y el texto introducido mediante la consola de comandos para realizar la acción deseada por el usuario, haciendo la aplicación una client line interface la cual genera a través de sus funcionalidades una web con los datos para poder visualizarlos de forma sencilla, son imágenes de las gráficas y datos ordenados.
Así el main se encarga de hacer la comprobación de la existencia de los directorios y archivos necesarios, como la comprobación de si se trata de un directorio o no para evitar que la aplicación reviente, la aplicación consta de 3 opciones comprobadas a través de esta clase, que son la opción de parsear y generar distintos documentos a partir de los csv dados, puede generar un csv parseado sin espacios, un json y el xml.

```kotlin
fun main(args: Array<String>) {
    val inicioEjecucion = System.currentTimeMillis()
    var currentExecution: Ejecucion

    if (args.size != 3 && args.size != 4) {
        println("Invalid number of arguments. [${args.size}]")
        println("""
            Possible arguments are:
            parser [directorio_origen] [directorio_destino]
            resumen [directorio_origen] [directorio_destino]
            resumen [distrito] [directorio_origen] [directorio_destino]
        """.trimMargin()
        )
        exitProcess(1)
    }

    if (args[0] != "parser" && args[0] != "resumen") {
        println("Invalid option")
        println(
            """
            Valid options are:
            parser
            resumen
        """.trimMargin()
        )
        exitProcess(2)
    }

    if (args.size == 3) {
        val origen = File(args[1])
        val destino = File(args[2])
        if (origen.exists()) {
            if (!origen.isDirectory) {
                println("$origen is not a directory.")
                exitProcess(3)
            }
        }
        if (destino.exists()) {
            if (!destino.isDirectory) {
                println("$destino is not a directory.")
                exitProcess(3)
            }
        }
    }

    if (args.size == 4) {
        val origen = File(args[2])
        val destino = File(args[3])
        if (origen.exists()) {
            if (!origen.isDirectory) {
                println("$origen is not a directory.")
                exitProcess(4)
            }
        }
        if (destino.exists()) {
```

```
if (destino.exists()) {
    if (!destino.isDirectory) {
        println("$destino is not a directory.")
        exitProcess(4)
    }
}

if (args[0] != "resumen") {
    println(
        """
        Option "resumen" is the only one which admits a total of 4 parameters.

        Valid arguments are:
        parser [directorio_origen] [directorio_destino]
        resumen [directorio_origen] [directorio_destino]
        resumen [distrito] [directorio_origen] [directorio_destino]
    """.trimIndent()
    )
    exitProcess(5)
}



(args.size == 3 && args[0] == "parser") {
 val parser = CSVParser(args[1], args[2])
 currentExecution = if (parser.parse() == 0) {
    Ejecucion("parser", inicioEjecucion, true)
 } else {
    Ejecucion("parser", inicioEjecucion, false)
 }
 BitacoraCreator.saveIntoBitacora(currentExecution)



(args.size == 3 && args[0] == "resumen") {
 val listResiduos = CSVReader.readCSVResiduos("${args[1]}${File.separator}modelo_residuos_2021.csv", ";")
 val listContenedores = CSVReader.readCSVContenedores("${args[1]}${File.separator}contenedores_varios.csv", ";")

 val processor = DataProcessor(listContenedores, listResiduos, inicioEjecucion, null, args[2])


(args.size == 4) {
 val listResiduos = CSVReader.readCSVResiduos("${args[2]}${File.separator}modelo_residuos_2021.csv", ";")
 val listContenedores = CSVReader.readCSVContenedores("${args[2]}${File.separator}contenedores_varios.csv", ";")

 val filteredResiduosList: List<Residuos> =
    listResiduos.stream().filter { x -> x.nombreDistrito.uppercase() == args[1].uppercase() }.toList()
 val filteredContenedoresList: List<Contenedor> =
    listContenedores.stream().filter { x -> x.distrito.uppercase() == args[1].uppercase() }.toList()
 if (filteredResiduosList.isEmpty() || filteredContenedoresList.isEmpty()) {
    println("District [${args[1]}] does not exist or there is no data related to it. Please select an existent district")
    exitProcess(6)
 }

 val processor = DataProcessor(filteredContenedoresList, filteredResiduosList, inicioEjecucion, args[1], args[3])
```