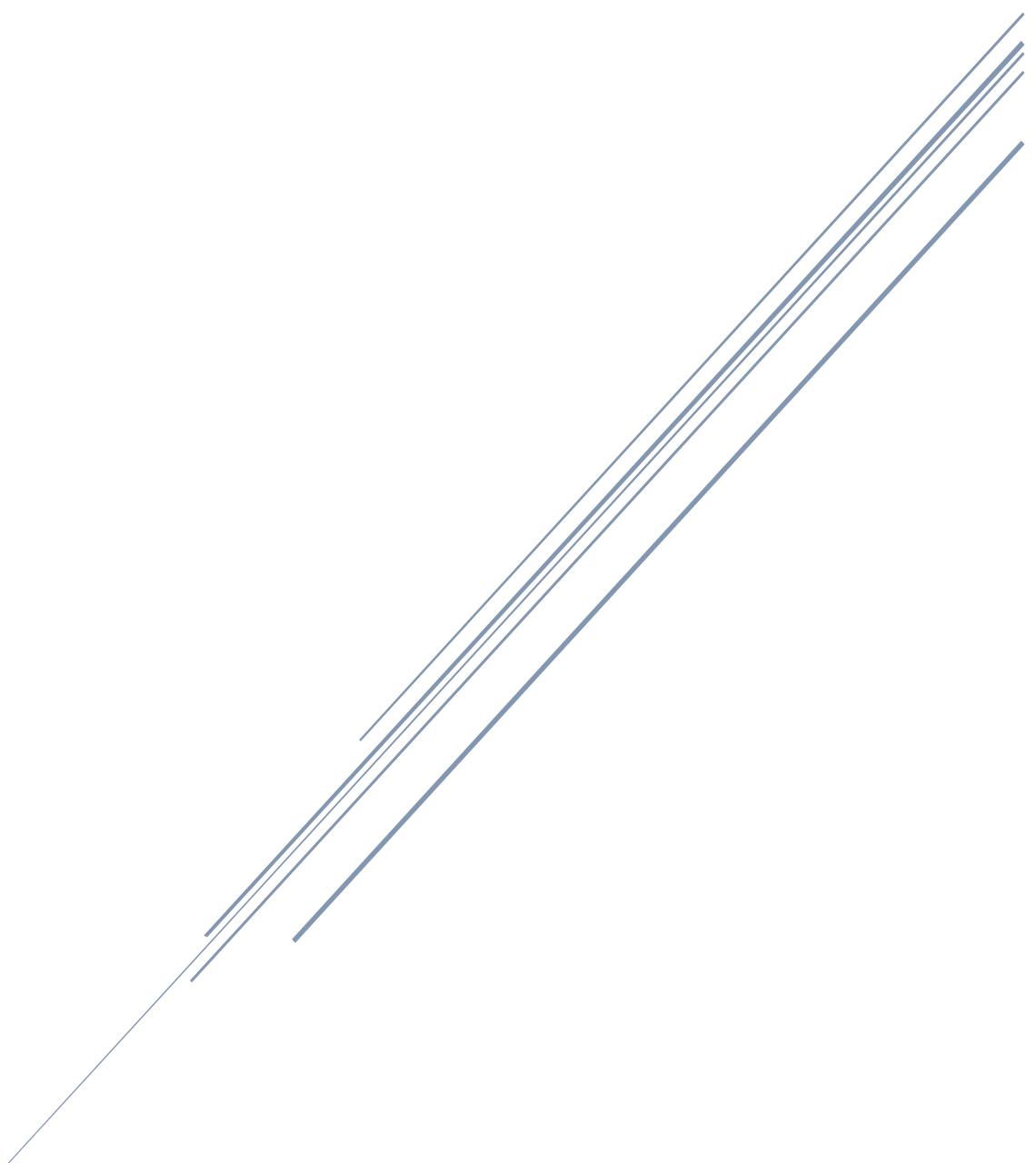


RECICLAJE Y LIMPIEZA DE MADRID

1^a práctica para Acceso a Datos sobre procesamiento de datos



IES Luis Vives
Mario Resa – Sebastián Mendoza

Contenido

Introducción	1
Elección del diseño.....	2
Análisis de proyecto	3
Bibliografía	24

Introducción

El manejo de datos y la filtración de la información es el día a día de todo programador, siendo una habilidad a la que se debe potenciar ya que el mercado laboral requiere de respuestas rápidas a la información que se almacena y a la que se accede en nuestros días. Es por ello que, en esta práctica, demostraremos que tenemos los conocimientos suficientes como para llevar a cabo una serie de filtrados de información mediante diversas tecnologías.

Elección del diseño

Una de las partes más importantes a la hora de empezar un proyecto es elegir cómo se va a llevar a cabo. Así pues, el elemento más básico por donde se empieza es la elección del lenguaje de programación que se va a utilizar.

Aunque el debate no fue muy extenso, nos decidimos a utilizar **Kotlin** como lenguaje base de nuestro proyecto. A pesar de que **Kotlin** es un lenguaje “nuevo” en relación a sus otras alternativas, se ha hecho un hueco en el mercado en un periodo corto de tiempo.

El motivo principal por el que elegimos **Kotlin** no solo recaía en las capacidades del mismo lenguaje, sino porque nos permitía utilizar la tecnología **DataFrame**. Esta tecnología proviene de Python, más concretamente, Panda. Básicamente **DataFrame** nos permite transformar las listas de información que tenemos en otras en las que se puede filtrar de forma más sencilla. Aún así, como toda tecnología en el mundo de la programación, tiene su dificultad a la hora de aprenderlo a usar o a implementarlo correctamente.

Como el planteamiento de la práctica nos pide transformar ficheros *.csv* en archivos *.json* y *.xml*, **DataFrame** nos venía de perlas a la hora de escribir datos de este tipo en *.json* y en nuevos *.csv*. En el caso del *.xml*, nos decidimos en utilizar una librería de serialización de xml para **Kotlin**.

Por otro lado, la práctica también necesita de gráficas para representar la información, por lo que en este caso utilizamos la librería *Lets-Plot* para **Kotlin**.

Y por último, pero no menos importante, decidimos utilizar *Logs* para imprimir por consola información relevante, así no tendríamos que echar mano de los *println()* de toda la vida. Esta implementación no sólo muestra un mensaje, sino también desde dónde sale el mensaje, haciendo posible la localización de algún problema o simplemente la localización de código en todo nuestro proyecto. También debemos señalar que para la documentación del código hemos utilizado *Dokka*, librería para **Kotlin** que nos permite exportar toda la documentación del programa a un sencillo *.html* con la información de todo nuestro proyecto.

Análisis de proyecto

Nuestro proyecto comienza con la configuración de del Gradle para poder utilizar las tecnologías anteriormente mencionadas:

```
plugins { this: PluginDependenciesSpecScope
    kotlin("jvm") version "1.7.10"
        // Para generar modelos de DataFrames
    id("org.jetbrains.kotlinx.dataframe") version "0.8.1"
        // Plugin para serializar
    kotlin("plugin.serialization") version "1.7.10"
        // Dokka Documentación Kotlin
    id("org.jetbrains.dokka") version "1.7.20"
        application
}
```

```
dependencies { this: DependencyHandlerScope
    testImplementation(kotlin("test"))
        // DataFrames de Kotlin Jetbrains
    implementation("org.jetbrains.kotlinx:dataframe:0.8.1")
        // Si quiero usar DateTime de Jetbrains Kotlin
    implementation("org.jetbrains.kotlinx:kotlinx-datetime:0.3.2")
        // Kotlin's serialization JSON
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.3.3")
        // Para hacer logs
    implementation("io.github.microutils:kotlin-logging-jvm:3.0.0")
    implementation("ch.qos.logback:logback-classic:1.4.1")
        // LetsPlot
    implementation("org.jetbrains.lets-plot:lets-plot-kotlin:3.2.0")
    implementation("org.jetbrains.lets-plot:lets-plot-image-export:2.3.0")
        // Serializa a XML con Serialization para jvm
        // https://github.com/pdvriese/xmlutil
    implementation("io.github.pdvriese.xmlutil:core-jvm:0.84.3")
    implementation("io.github.pdvriese.xmlutil:serialization-jvm:0.84.3")
    testImplementation("org.junit.jupiter:junit-jupiter:5.8.1")
        // Dokka Documentación Kotlin
    dokkaHtmlPlugin("org.jetbrains.dokka:kotlin-as-java-plugin:1.7.20")
}
```

```

tasks {
    val fatJar = register<Jar>("fatJar") {
        dependsOn.addAll(listOf("compileJava", "compileKotlin", "processResources")) // We need this for Gradle optimization to work
        duplicatesStrategy = DuplicatesStrategy.EXCLUDE
        manifest { attributes(mapOf("Main-Class" to application.mainClass)) } // Provided we set it up in the application plugin configuration
        val sourcesMain = sourceSets.main.get()
        val contents = configurations.runtimeClasspath.get()
            .map { if (it.isDirectory) it else zipTree(it) } +
            sourcesMain.output
        from(contents)
    }
    build {
        dependsOn(fatJar) // Trigger fat jar creation during build
    }
}
tasks.withType<Jar> {
    manifest {
        attributes["Main-Class"] = "MainKt"
    }
}

```

Configurado el Gradle, lo primero es revisar las cabeceras de los .csv ya que son diferentes. El objetivo de esto es la creación de las clases modelo:

```

@DataSchema
@Serializable
@SerializedName("Contenedores")
data class Contenedores(
    @XmlElement(true)
    val codIntCont: String,
    @XmlElement(true)
    val tipoCont: String,
    @XmlElement(true)
    val modeloCont: String,
    @XmlElement(true)
    val descModeloCont: String,
    @XmlElement(true)
    val cantidadCont: Int,
    @XmlElement(true)
    val loteCont: String,
    @XmlElement(true)
    val distritoCont: String,
)

```

```
@XmlElement(true)
    val barrioCont: String?,
    @XmlElement(true)
    val viaCont: String,
    @XmlElement(true)
    val nomViaCont: String,
    @XmlElement(true)
    val numCalleCont: String,
    @XmlElement(true)
    val coorXCont: String,
    @XmlElement(true)
    val coorYCont: String,
    @XmlElement(true)
    val longiCont: String,
    @XmlElement(true)
    val latiCont: String,
    @XmlElement(true)
    val dirCompletaCont: String
```

```
@DataSchema
@Serializable
@SerializedName("Residuos")
data class Residuos(
    @XmlElement(true)
    val anioResi: String,
    @XmlElement(true)
    val mesResi: String,
    @XmlElement(true)
    val loteResi: String,
    @XmlElement(true)
    val tipoResi: String,
    @XmlElement(true)
    val codDistritoResi: String,
    @XmlElement(true)
    val nomDistritoResi: String,
    @XmlElement(true)
    val toneladasResi: Double
)
```

Teniendo los modelos hechos, el siguiente paso fue crear una función de lectura de .csv para cada archivo. La creación de estas funciones en las clases modelo lo decidimos para tener más control de los campos de la información y para poder arreglar los campos que necesitáramos, ya que algunos campos iban a funcionar como Double, por ejemplo, pero perfectamente podríamos haber hecho la lectura de los archivos en las clases que controlaran las filtraciones:

```
fun loadCsvCont(csvFile: File): List<Contenedores> {
    val contenedores: List<Contenedores> = csvFile.readLines() List<String>
        .drop( n: 1 )
        .map { it.split( ...delimiters: ";" ) } List<List<String>>
        .map { it: List<String>
            it.map { campo → campo.trim() }
            Contenedores(
                codIntCont = it[0],
                tipoCont = it[1],
                modeloCont = it[2],
                descModeloCont = it[3],
                cantidadCont = it[4].toInt(),
                loteCont = it[5],
                distritoCont = arregloEspacios(it[6]),
                barrioCont = it[7],
                viaCont = it[8],
                nomViaCont = it[9],
                numCalleCont = it[10],
                coorXCont = it[11],
                coorYCont = it[12],
                longiCont = it[13],
                latiCont = it[14],
                dirCompletaCont = it[15]
            )
        }
    return contenedores
}
```

```
fun arregloEspacios(dato: String): String {
    var nuevo = dato.replace("\u00a0".toRegex(), " ")
    if (dato.contains( other: " - ")){
        nuevo = Normalizer.normalize(dato, Normalizer.Form.NFD).replace( oldValue: " - ", newValue: "-")
    }
    return nuevo.uppercase()
}
```

```
fun loadCsvResi(csvFile: File): List<Residuos> {
    val residuos: List<Residuos> = csvFile.readLines() List<String>
        .drop( n: 1)
        .map { it.split( ...delimiters: ";" ) } List<List<String>>
        .map { it: List<String>
            it.map { campo → campo.trim() }
            Residuos(
                anioResi = it[0],
                mesResi = it[1],
                loteResi = it[2],
                tipoResi = it[3],
                codDistritoResi = it[4],
                nomDistritoResi = igualarString(it[5]),
                toneladasResi = punto(it[6])
            )
        }
    return residuos
}
```

```
fun punto(dato: String): Double {
    val nuevo: String = dato.replace( oldValue: ", ", newValue: ".")
    return nuevo.toDouble()
}
```

```
fun igualarString(dato: String): String {
    var nuevo: String = Normalizer.normalize(dato, Normalizer.Form.NFD).replace("[^\\p{ASCII}]".toRegex(), replacement: "")
    if (dato.contains( other: " - ")){
        nuevo = Normalizer.normalize(dato, Normalizer.Form.NFD).replace( oldValue: " - ", newValue: "-")
    }
    return nuevo.uppercase()
}
```

A este punto del proyecto decidimos crear dos clases: la clase Main y una clase controller que comprueba los directorios donde están los .cvs.

El diseño del Main se hizo en función de cómo trabajaría el .jar, pues éste recibiría parámetros y realizaría un procedimiento u otro. El resultado fue el siguiente:

```
fun main(args: Array<String>) {
    logger.debug { "Ejecutando aplicación" }
    when (args.size) {
        0, 1, 2 ->
            logger.debug { "Sin parametros o datos erroneos: Vuelve a ejecutar el programa con una opcion, una carpeta de origen de los datos y otra de destino" }

        3 -> {
            when (args[0].lowercase()) {
                "parser" -> {
                    DirController.init(args[1], args[2])
                    ParserController.init(args[1], args[2])
                }

                "resumen" -> {
                    DirController.init(args[1], args[2])
                    ResumenController.init(args[1], args[2])
                }

                else -> {
                    logger.debug { "Opcion erronea: Vuelve a ejecutar el programa con una opción valida, una carpeta de origen de los datos y otra de destino" }
                }
            }
        }
    }
}

4 -> {
    when (args[0].lowercase()) {
        "resumen" -> {
            DirController.init(args[2], args[3])
            DistritoController.init(args[1], args[2], args[3])
        }

        else -> {
            logger.debug { "Parametros erroneos: Vuelve a ejecutar el programa con una opcion valida, una carpeta de origen de los datos y otra de destino" }
        }
    }
}

else -> {
    logger.debug { "Parametros erroneos: Vuelve a ejecutar el programa con una opcion valida, una carpeta de origen de los datos y otra de destino" }
}

logger.debug { "Fin de la aplicación" }
```

Como se puede ver en la imagen, las funciones trabajan según el tamaño del *Array<String>* y qué contenido tiene cada array.

Para poder trabajar con los archivos .csv, decidimos que debíamos crear una clase que controlara las carpetas y los archivos, pues el proyecto funcionaría introduciendo una opción, una dirección de origen y otra de destino.

En nuestro caso, decidimos que este proyecto trabajaría exclusivamente con los archivos .csv de la Comunidad de Madrid y no otros, pues la lógica que nos guiaba era que no tenía sentido permitir que leyera archivos diferentes, pero con la misma extensión.

Así pues, la clase DirController quedó así:

```
fun init(dirOrigen: String, dirDestino: String) {
    val workingDir: String = System.getProperty("user.dir")
    val dirOrigenPath = Paths.get(dirOrigen)
    val dirDestinoPath = Paths.get(dirDestino)
    val fileContOrigen = Paths.get(first: dirOrigen + File.separator + "contenedores_varios.csv")
    val fileResiOrigen = Paths.get(first: dirOrigen + File.separator + "modelo_residuos_2021.csv")
    val fileBitacoraPath = Paths.get(first: workingDir + File.separator + "bitacora")

    if (Files.isDirectory(fileBitacoraPath) && Files.exists(fileBitacoraPath)) {
        logger.debug { "Carpeta de bitacora comprobada... OK" }
    } else {
        logger.debug { "Carpeta de bitacora no existe. Creando..." }
        Files.createDirectory(fileBitacoraPath)
        logger.debug { "Carpeta de bitacora creada..." }
    }

    if (Files.isDirectory(dirOrigenPath) && Files.exists(dirOrigenPath) && Files.exists(fileContOrigen) && Files.exists(
        fileResiOrigen
    )) {
        logger.debug { "Carpeta de origen comprobada... OK" }
    } else {
        logger.debug { "Proporciona una carpeta de origen valida... Fin de la aplicacion" }
        exitProcess(status: 0)
    }
}

if (Files.isDirectory(dirDestinoPath) && Files.exists(dirDestinoPath)) {
    logger.debug { "Carpeta de destino comprobada... OK" }
} else {
    logger.debug { "No existe la carpeta. Creando..." }
    Files.createDirectory(dirDestinoPath)
    Files.isWritable(dirDestinoPath)
    logger.debug { "Carpeta de destino creada..." }
}
```

En las imágenes podemos ver que, al dar archivos o carpetas diferentes, el programa avisa del error y se cierra, para que, en su siguiente ejecución, se den los parámetros correctos.

La primera opción que nos pide el trabajo es la transformación de los .csv en otros archivos, siendo `.json`, `.xml` y `.csv`.

Así pues, creamos la clase `ParserController`, la cual manejará las funciones que comprobarán que los datos y las carpetas ofrecidas son correctas. En dicha clase disponemos de dos funciones, siendo `init()` la primera y `parserCsv()` la segunda. El diseño de dichas funciones es el siguiente:

```
fun init(dirOrigen: String, dirDestino: String): Boolean {
    val workingDir: String = System.getProperty("user.dir")
    val fileBitacoraPath = Paths.get(first: workingDir + File.separator + "bitacora")

    if (Files.isDirectory(fileBitacoraPath) && Files.exists(fileBitacoraPath)) {
        logger.debug { "Carpeta de bitacora comprobada... OK" }
    } else {
        logger.debug { "Carpeta de bitacora no existe. Creando..." }
        Files.createDirectory(fileBitacoraPath)
        logger.debug { "Carpeta de bitacora creada..." }
    }

    var proceso: Boolean
    try {
        logger.debug { "Creando archivos..." }
        val csvContenedores = dirOrigen + fs + "contenedores_varios.csv"
        val csvResiduos = dirOrigen + fs + "modelo_residuos_2021.csv"
        val destinoPath = dirDestino + fs
    }
}
```

```
//Lectura de csv
val cont by lazy { loadCsvCont(File(csvContenedores)) }
val resi by lazy { loadCsvResi(File(csvResiduos)) }

val tiempo = measureTimeMillis {
    parserCsv(cont, resi, destinoPath)
}
createInforme(tiempo.toString())
logger.debug { "Tiempo: $tiempo ms" }
proceso = true

} catch (e: IOException) {
    exito = false
    logger.error(e.message)
    proceso = false
    createInforme( tiempo: "0")
}
return proceso
}
```

```

private fun parserCsv(cont: List<Contenedores>, resi: List<Residuos>, destino: String) {
    val dfCont by lazy { cont.toDataFrame() }
    val dfResi by lazy { resi.toDataFrame() }
    //Seleccion de columnas
    val contNuevoCsv = dfCont.select { it.tipoCont and it.cantidadCont and it.distritoCont }
    val resiNuevoCsv = dfResi.select { it.mesResi and it.tipoResi and it.nomDistritoResi and it.toneladasResi }
    //Creacion CSV nuevo
    contNuevoCsv.writeCSV(File( pathname: destino + "contenedoresCsv.csv"), CSVFormat.DEFAULT.withDelimiter( delimiter: ';'))
    resiNuevoCsv.writeCSV(File( pathname: destino + "residuosCsv.csv"), CSVFormat.DEFAULT.withDelimiter( delimiter: ';'))
    //Creacion de JSON
    contNuevoCsv.writeJson(File( pathname: destino + "contenedoresJson.json"), prettyPrint = true)
    resiNuevoCsv.writeJson(File( pathname: destino + "residuosJson.json"), prettyPrint = true)
    //Creacion de XML
    val xml = XML { indentString = " " }
    val contenedoresXml = File( pathname: destino + "contenedoresXML.xml")
    contenedoresXml.writeText(xml.encodeToString(cont))
    val residuosXml = File( pathname: destino + "residuosXml.xml")
    residuosXml.writeText(xml.encodeToString(resi))
}

```

A colación con lo anterior, en este punto mencionaremos los informes. La práctica en cuestión pide que se genere un informe sobre el resultado de los procedimientos, empezando entonces por el del ParserController.

Para poder hacer dichos informes (los cuales tienen que ser en formato .xml), primero diseñamos una nueva clase modelo llamada *Informe*:

```

@Serializable
@SerializedName("Informe")
data class Informe(
    var id: String,
    var createdAt: String,
    @XmlElement(true)
    var opcion: String,
    @XmlElement(true)
    var exito: String,
    @XmlElement(true)
    var tiempo: String
) {
    ▾ Mario
    companion object
}

```

```
fun Informe.Companion.writeToXmlFile(informe: Informe, xmlFile: File) {
    logger.debug { "Escribiendo informe..." }
    val xml = XML { indentString = " " }
    xmlFile.appendText(xml.encodeToString(informe))
    xmlFile.appendText( text: "\n")
    logger.debug { "Informe realizado con exito" }
}
```

En la última imagen podemos ver la función que escribirá dichos informes.

“Parseados” los archivos, lo siguiente fue realizar la siguiente operación de la práctica: el resumen general. Dicho resumen es una concatenación de filtraciones sobre datos generalizados en todos los distritos disponibles de los .csv.

Dicha operación la encapsulamos en una clase object llamada ResumenController. En dicha clase creamos dos funciones, **init()** y **procesosFiltrados()**. Aunque en las siguientes imágenes aparecen otras funciones, se explicarán en el proceso:

```
fun init(dirOrigen: String, dirDestino: String) {
    val csvContenedores = dirOrigen + fs + "contenedores_varios.csv"
    val csvResiduos = dirOrigen + fs + "modelo_residuos_2021.csv"
    val destinoPath = dirDestino + fs

    //Lectura de csv
    val cont by lazy { loadCsvCont(File(csvContenedores)) }
    val resi by lazy { loadCsvResi(File(csvResiduos)) }

    val tiempo = measureTimeMillis {
        procesoFiltrados(cont, resi)
    }
    createInforme(tiempo.toString())
    logger.debug { "Tiempo: $tiempo ms" }

    val fecha = LocalDateTime.now().format(DateTimeFormatter.ofPattern(pattern: "dd MMM yyyy, HH:mm:ss"))
    logger.debug { fecha }

    createHtmlResumen(destinoPath, tiempo.toString(), fecha)
    logger.debug { "Resumen HTML realizado" }
}
```

Como se puede ver en la imagen anterior, aparte de la lectura de los archivos .csv, están las funciones de creación del informe y la creación del .html. Estas funciones trabajarán con la información obtenida en las filtraciones que veremos en las siguientes imágenes.

- Número de contenedores de cada tipo que hay en cada distrito

```
// FILTRADOS
logger.debug { "Número de contenedores de cada tipo que hay en cada distrito" }
numTipoContxDistrito = dfCont.groupBy { it.distritoCont.rename( newName: "Distrito") }
    .aggregate { this: AggregateGroupedDsl<Contenedores>  it: AggregateGroupedDsl<Contenedores>
        count { it.tipoCont == "RESTO" } into "Restos"
        count { it.tipoCont == "PAPEL-CARTON" } into "Papel-Carton"
        count { it.tipoCont == "ORGANICA" } into "Organica"
        count { it.tipoCont == "ENVASES" } into "Envases"
        count { it.tipoCont == "VIDRIO" } into "Vidrio"
    }.sortBy { it["Distrito"] }
```

- Media de contenedores de cada tipo por distrito

```
logger.debug { "Media de contenedores de cada tipo por distrito" }
mediaTipoContxDistrito =
    dfCont.groupBy { it.distritoCont.rename( newName: "Distrito") and it.tipoCont.rename( newName: "Contenedores") }
        .aggregate { mean { it.cantidadCont } into "Media" }.sortBy { it["Distrito"] }
```

- Media de toneladas anuales de recogidas por cada tipo de basura agrupadas por distrito

```
logger.debug { "Media de toneladas anuales de recogidas por cada tipo de basura agrupadas por distrito" }
mediaTonResiDistritos =
    dfResi.groupBy { it.nomDistritoResi.rename( newName: "Distrito") and it.tipoResi.rename( newName: "Tipo") }
        .aggregate { mean { it.toneladasResi } into "Media" }.sortBy { it["Distrito"] }
```

- Máximo, mínimo, media y desviación de toneladas anuales de recogidas por cada tipo de basura agrupadas por distrito

```
logger.debug { "Maximo, minimo, media y desviacion de toneladas anuales de recogidas por cada tipo de basura agrupadas por distrito" }
maxToneladasDistrito =
    dfResi.groupBy { it.nomDistritoResi.rename( newName: "Distrito") and it.tipoResi.rename( newName: "Tipo") }
        .aggregate { this: AggregateGroupedDsl<Residuos>  it: AggregateGroupedDsl<Residuos>
            max { it.toneladasResi } into "Max"
            min { it.toneladasResi } into "Min"
            mean { it.toneladasResi } into "Media"
            std { it.toneladasResi } into "Desviacion"
        }.sortBy { it["Distrito"] }
```

- Suma de las toneladas recogidas anualmente por distrito

```
logger.debug { "Suma de las toneladas recogidas anual por distrito" }
sumToneladasDistrito =
    dfResi.groupBy { it.nomDistritoResi.rename( newName: "Distrito" ) }
        .aggregate { sum { it.toneladasResi } into "TotalToneladas" }.sortBy { it["Distrito"] }
```

- Por cada distrito obtener para cada tipo de residuo la cantidad recogida

```
logger.debug { "Por cada distrito obtener para cada tipo de residuo la cantidad recogida" }
toneladasDistrito = dfResi.groupBy { it.nomDistritoResi.rename( newName: "Distrito" ) and it.tipoResi.rename( newName: "Tipo" ) }
    .aggregate { this: AggregateGroupedDsl<Residuos> -> it: AggregateGroupedDsl<Residuos>
        sum { it.toneladasResi } into "TotalToneladas"
    }.sortBy { it["Distrito"] }
```

A parte de estas consultas, también se han realizado gráficas a partir de alguna de ellas. Es aquí donde utilizamos la librería *Lets-Plot* para poder generar gráficas con las consultas **DataFrames**.

- Total de contenedores por distrito

```
val numContenedores =
    dfCont.groupBy { it.distritoCont }.aggregate { count() into "contenedores" }.sortBy { it.distritoCont }
var fig: Plot = letsPlot(data = numContenedores.toMap()) + geomBar(
    stat = Stat.identity, alpha = 0.8
) { this: BarMapping
    x = "distritoCont"; y = "contenedores"
} + labs(
    x = "Distrito", y = "Contenedores", title = "Total contenedores por distrito"
)
ggsave(fig, filename: "ContenedoresPorDistrito.png")
```

- Gráfico de media de toneladas mensuales de recogida de basura por distrito

```
val mediaToneladas =
    dfResi.groupBy { it.mesResi.rename( newName: "Mes" ) and it.nomDistritoResi.rename( newName: "Distrito" ) }
        .aggregate { mean { it.toneladasResi } into "Media" }
fig = letsPlot(data = mediaToneladas.toMap()) +
    geomTile(height = 0.9, width = 0.9) { x = "Distrito"; y = "Mes"; fill = "Media" } +
    theme(panelBackground = elementBlank(), panelGrid = elementBlank() + scaleFillGradient(
        low = "#00FFE5",
        high = "#006D63"
    ) + ggtitle( title: "Media de toneladas por distrito y mes" ) + ggszie( width: 900, height: 700 )
ggsave(fig, filename: "ToneladasPorDistrito.png")
```

Después de las filtraciones, se realiza el informe:

```
private fun createInforme(tiempo: String) {
    val informe = Informe(
        UUID.randomUUID().toString(),
        LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME).toString(),
        opcion: "Resumen Global",
        exito: "Proceso Exitoso",
        tiempo: "$tiempo milisegundos"
    )
    Informe.writeToXmlFile(informe, File( pathname: "bitacora${fs}bitacora.xml"))
}
```

Además, generamos también el *.html* donde se verán todas las filtraciones y las gráficas realizadas. Para hacer dicho documento, la opción que elegimos fue la de crear un *Template* directamente en un String, el cual nos permitía introducir información directa de nuestras filtraciones al documento. El resultado fue el siguiente:

```
private fun createHtmlResumen(dirDestino: String, tiempo: String, fecha: String) {
    val workingDir: String = System.getProperty("user.dir")
    val pathPlot1 = File( pathname: "${workingDir}${fs}lets-plot-images${fs}ContenedoresPorDistrito.png")
    val pathPlot2 = File( pathname: "${workingDir}${fs}lets-plot-images${fs}ToneladasPorDistrito.png")
    val fileHtml = File( pathname: dirDestino + "ResumenGlobal.html")
    val fileCss = File( pathname: dirDestino + "ResumenGlobal.css")

    var data1 = ""
    for (i in numTipoContXDistrito) {
        data1 += """<tr><td>${i["Distrito"]}</td><td>${i["Restos"]}</td><td>${i["Papel-Carton"]}</td>
        <td>${i["Organica"]}</td><td>${i["Envases"]}</td><td>${i["Vidrio"]}</td></tr>""".trimIndent()
    }

    var data2 = ""
    for (i in mediaTipoContXDistrito) {
        data2 += """<tr><td>${i["Distrito"]}</td><td>${i["Contenedores"]}</td><td>${i["Media"]}</td>
        </tr>""".trimIndent()
    }

    var data3 = ""
    for (i in mediaTonResiDistritos) {
        data3 += """<tr><td>${i["Distrito"]}</td><td>${i["Tipo"]}</td><td>${i["Media"]}</td></tr>
        """.trimIndent()
    }
}
```

```

var data4 = ""
for (i in maxToneladasDistrito) {
    data4 += """<tr><td>${i["Distrito"]}</td><td>${i["Tipo"]}</td><td>${i["Max"]}</td><td>${i["Min"]}</td>
    <td>${i["Media"]}</td><td>${i["Desviacion"]}</td></tr>""".trimIndent()
}
var data5 = ""
for (i in sumToneladasDistrito) {
    data5 += """<tr><td>${i["Distrito"]}</td><td>${i["TotalToneladas"]}</td></tr>"""
}
var data6 = ""
for (i in toneladasDistrito) {
    data6 += """<tr><td>${i["Distrito"]}</td><td>${i["Tipo"]}</td><td>${i["TotalToneladas"]}</td></tr>"""
}

```

```

fileHtml.writeText(
"""
<!DOCTYPE html>
<html>
    <head><link rel="stylesheet" href="$fileCss"></head>
    <body>
        <div class="container">
            <div class="cabecera">
                <h1>Resumen de recogidas de basura y reciclaje en Madrid</h1>
                <h4>Fecha y Hora: $fecha</h4>
                <h4>Autores: Mario Resa y Sebastian Mendoza</h4>
            </div>
            <div class="resumen">
                <h3>Número de contenedores de cada tipo que hay en cada distrito</h3>
                <table border="1">
                    <tr><th>Distrito</th><th>Restos</th><th>Papel-Carton</th><th>Organica</th><th>Envases</th><th>Vidrio</th></tr>
                    $data1
                </table>
                <h3>Media de contenedores de cada tipo que hay en cada distrito</h3>
                <table border="1">
                    <tr><th>Distrito</th><th>Contenedor</th><th>Media</th></tr>
                    $data2
                </table>
            </div>
        </div>
    </body>
</html>

```

```

</table>
<h3>Total de contenedores por distrito</h3>

<h3>Media de toneladas anuales de recogidas por cada tipo de basura agrupadas por distrito</h3>
<table border="1">
    <tr><th>Distrito</th><th>Tipo</th><th>Media</th></tr>
    $data3
</table>
<h3>Media de toneladas mensuales de recogida de basura por distrito</h3>

<h3>Maximo, minimo , media y desviacion de toneladas anuales de recogidas agrupadas por distrito</h3>
<table border="1">
    <tr><th>Distrito</th><th>Tipo</th><th>Maximo</th><th>Minimo</th><th>Media</th><th>Desviacion</th></tr>
    $data4
</table>
<h3>Suma de todo lo recogido en un año por distrito</h3>
<table border="1">
    <tr><th>Distrito</th><th>Total Toneladas</th></tr>
    $data5

```

```

        </table>
        <h3>Por cada distrito obtener para cada tipo de residuo la cantidad recogida</h3>
        <table border="1">
        <tr><th>Distrito</th><th>Tipo</th><th>Total Toneladas</th></tr>
        $data6
        </table>
        <h3>Tiempo de generacion: $tiempo ms</h3>
    </div>
</div>
</body>
</html>
""".trimIndent()
)

```

A parte del archivo que será el *.html*, también creamos la hoja de estilos *.css*:

```

fileCss.writeText(
"""
body { margin: 0;
background-color: #D7DBDD}

.cabecera {
background-color: #CACFD2;
width: 100%;
}

.container, .resumen {
display: flex;
align-items: center;
justify-content: center;
flex-direction: column
}

h3,td, .cabecera { text-align: center
}

th { background-color: #B3E6FF }
tr:hover { background-color: #D7DBDD }

table { margin: 0;
background-color: #FBFCFC}

""".trimIndent()
)

```

Finalmente, la práctica pide la operación anterior, pero filtrando por un distrito dado. Relativamente hablando, guarda una similitud enorme con la clase anterior. El único cambio que hacemos en la función que realiza los filtrados, pues es allí donde comprobamos que el distrito introducido por parámetros existe o no. El código es el siguiente:

```
private fun procesoFiltrados(distrito: String, cont: List<Contenedores>, resi: List<Residuos>) {
    logger.debug { "Inicio de filtrado por distrito: $distrito" }
    val dfCont by lazy { cont.toDataFrame() }
    val dfResi by lazy { resi.toDataFrame() }

    logger.debug { "Comprobando nombre del distrito..." }
    val dfDistritos = dfCont.count { it.distritoCont == distrito }
    if (dfDistritos == 0) {
        logger.debug { "No existe el distrito" }
        exito = false
        createInforme(distrito, tiempo: "0")
        exitProcess(status: 0)
    }
}
```

- Número de contenedores de cada tipo, distrito específico

```
logger.debug { "Número de contenedores de cada tipo, distrito específico" }
numTipoContXDistrito =
    dfCont.filter { it.distritoCont == distrito }
        .aggregate { this: AggregateGroupedDsl<Contenedores> it: AggregateGroupedDsl<Contenedores>
            count { it.tipoCont == "RESTO" } into "Restos"
            count { it.tipoCont == "PAPEL-CARTON" } into "Papel-Carton"
            count { it.tipoCont == "ORGANICA" } into "Organica"
            count { it.tipoCont == "ENVASES" } into "Envases"
            count { it.tipoCont == "VIDRIO" } into "Vidrio"
        }
```

- Total de toneladas recogidas en este distrito por residuos

```
logger.debug { "Total de toneladas recogidas en este distrito por residuos" }
totalTonResiDistrito =
    dfResi.filter { it.nomDistritoResi == distrito }.groupBy { it.tipoResi.rename( newName: "Tipo") }
        .aggregate { this: AggregateGroupedDsl<Residuos> it: AggregateGroupedDsl<Residuos>
            sum { it.toneladasResi } into "Total"
        }.sortBy { it["Total"].desc() }
```

- Máximo, mínimo, media y desviación

```
logger.debug { "Maximo, minimo, media y desviacion" }
operacionesToneladas = dfResi.filter { it.nomDistritoResi == distrito } DataFrame<Residuos>
    .groupBy { it.tipoResi.rename( newName: "Tipo" ) } GroupBy<Residuos, Residuos>
    .aggregate { this: AggregateGroupedDsl<Residuos> it: AggregateGroupedDsl<Residuos>
        maxBy{it.toneladasResi}["mesResi"] into "MesMax"
        max{it.toneladasResi} into "Max"
        minBy { it.toneladasResi }["mesResi"] into "MesMin"
        min(it.toneladasResi) into "Min"
        mean(it.toneladasResi) into "Media"
        std(it.toneladasResi) into "Desviacion"
    }
```

De la misma manera que en el caso anterior, también se han generado gráficas:

- Barras (Total Toneladas X Residuo) en determinado Distrito

```
var fig: Plot = letsPlot(data = totalTonResiDistrito.toMap()) + geomBar(
    stat = Stat.identity, alpha = 0.8
) { this: BarMapping
    X = "Tipo"; y = "Total"
} + labs(
    x = "Residuos", y = "Toneladas", title = "Total Toneladas por Residuo en $distrito"
)
ggsave(fig, filename: "ToneladasPorResiduo${distrito}.png")
```

- Gráfico (Maximo, minimo y media por meses en dicho distrito)

```
val operacionGrafica = dfResi.filter { it.nomDistritoResi == distrito } DataFrame<Residuos>
    .groupBy { it.mesResi into "Meses" } GroupBy<Residuos, Residuos>
    .aggregate { this: AggregateGroupedDsl<Residuos> it: AggregateGroupedDsl<Residuos>
        max(it.toneladasResi) into "Max"
        min(it.toneladasResi) into "Min"
        mean(it.toneladasResi) into "Media"
    }
```

```

fig = letsPlot(data = operacionGrafica.toMap()) + geomBar(
    stat = Stat.identity,
    alpha = 0.8,
    fill = Color.BLACK,
) { this: BarMapping
    x = "Meses"; y = "Max"
} + geomBar(
    stat = Stat.identity,
    alpha = 0.8,
    fill = Color.RED
) { this: BarMapping
    x = "Meses"; y = "Media"
} + geomBar(
    stat = Stat.identity,
    alpha = 0.8,
    fill = Color.PACIFIC_BLUE
) { this: BarMapping
    x = "Meses"; y = "Min"
} + labs(
    x = "Meses",
    y = "Operaciones",
    title = "Maximo, media y minimo para $distrito por meses"
)
ggsave(fig, filename: "Operaciones${distrito}.png")

```

Después de las filtraciones se genera el informe:

```

private fun createInforme(distritoMain: String, tiempo: String) {
    var exitoString = "Proceso exitoso"
    if (!exito) {
        exitoString = "Proceso fallido"
    }
    val informe = Informe(
        UUID.randomUUID().toString(),
        LocalDateTime.now().toString(),
        opcion: "Resumen $distritoMain",
        exitoString,
        tiempo: "$tiempo milisegundos"
    )
    Informe.writeToXmlFile(informe, File( pathname: "bitacora${fs}bitacora.xml"))
}

```

Y como última función, la del informe en .html:

```
private fun createHtmlDistrito(distrito: String, dirDestino: String, tiempo: String, fecha: String) {
    val workingDir: String = System.getProperty("user.dir")
    val pathPlot1 = File( pathname: "${workingDir}${fs}lets-plot-images${fs}ToneladasPorResiduo${distrito}.png")
    val pathPlot2 = File( pathname: "${workingDir}${fs}lets-plot-images${fs}Operaciones${distrito}.png")
    val fileHtml = File( pathname: dirDestino + "resumen_${distrito}.html")
    val fileCss = File( pathname: dirDestino + "resumen_distrito.css")

    val data1 =
        """<tr><td>${numTipoContXDistrito[0]}</td><td>${numTipoContXDistrito[1]}</td>
        <td>${numTipoContXDistrito[2]}</td><td>${numTipoContXDistrito[3]}</td>
        <td>${numTipoContXDistrito[4]}</td></tr>""".trimIndent()
    var data2 = ""
    for (i in totalTonResiDistrito) {
        data2 += """<tr><td>${i["Tipo"]}</td><td>${i["Total"]}</td></tr>"""
    }
    var data3 = ""
    for (i in operacionesToneladas) {
        data3 += """<tr><td>${i["Tipo"]}</td><td>${i["MesMax"]}</td><td>${i["Max"]}</td>
        <td>${i["MesMin"]}</td><td>${i["Min"]}</td><td>${i["Media"]}</td>
        <td>${i["Desviacion"]}</td></tr>""".trimIndent()
    }
}
```

```
fileHtml.writeText(
"""
<!DOCTYPE html>
<html>
    <head><link rel="stylesheet" href="$fileCss"></head>
    <body>
        <div class="container">
            <div class="cabecera">
                <h1>Resumen de recogidas de basura y reciclaje en $distrito</h1>
                <h4>Fecha y Hora: $fecha</h4>
                <h4>Autores: Mario Resa y Sebastian Mendoza</h4>
            </div>
            <div class="resumen">
                <h3>Número de contenedores de cada tipo que hay en este distrito</h3>
                <table border="1">
                    <tr><th>Restos</th><th>Papel-Carton</th><th>Organica</th><th>Envases</th><th>Vidrio</th></tr>
                $data1
                </table>
                <h3>Total de toneladas recogidas en este distrito por residuo</h3>
                <table border="1">
                    <tr><th>Tipo</th><th>Total</th></tr>
                $data2

```

```

                </table>
                <h3>Total de toneladas por residuo en este distrito</h3>
                
                <h3>Máximo, mínimo, media y desviación por mes por residuo en este distrito</h3>
                <table border="1">
                    <tr><th>Tipo</th><th>MesMax</th><th>Max</th><th>MesMin</th><th>Min</th><th>Media</th><th>Desviacion</th></tr>
                $data3
                </table>
                <h3>Máximo, mínimo y media por meses en dicho distrito</h3>
                
                <h3>Tiempo de generación: $tiempo ms</h3>
            </div>
        </div>
    </body>
</html>
""".trimIndent()
```

```
fileCss.writeText(  
    """ body { margin: 0;  
        background-color: #D7DBDD}  
  
        .cabecera {  
            background-color: #CACFD2;  
            width: 100%;  
        }  
  
        .container, .resumen {  
            display: flex;  
            align-items: center;  
            justify-content: center;  
            flex-direction: column  
        }  
  
        h3,td, .cabecera { text-align: center  
    }  
  
    th { background-color: #B3E6FF }  
    tr:hover { background-color: #D7DBDD }  
  
    table { margin: 0;  
        background-color: #FBFCFC}  
  
    """.trimIndent()  
)  
}
```

Al hacer *build* en Gradle, se generará un archivo *.jar* llamado *Basura.jar* totalmente operativa.

Para terminar la documentación, presentaremos un ejemplo de ejecución para mostrar el resultado final en el informe *.html*:

Resumen de recogidas de basura y reciclaje en SAN BLAS-CANILLEJAS

Fecha y Hora: 19 oct 2022, 22:52:31

Autores: Mario Resa y Sebastian Mendoza

Numero de contenedores de cada tipo que hay en este distrito

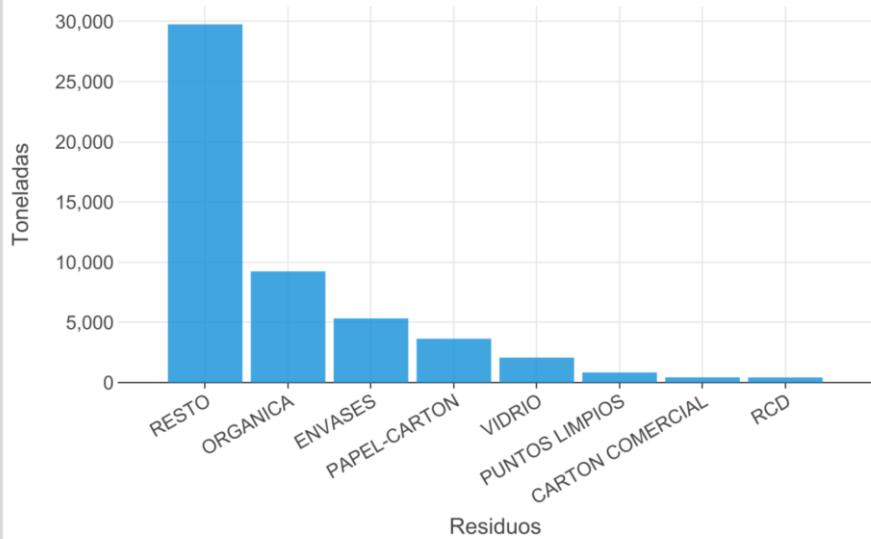
Restos	Papel-Carton	Organica	Envases	Vidrio
439	505	344	360	452

Total de toneladas recogidas en este distrito por residuo

Tipo	Total
RESTO	29774.36
ORGANICA	9244.7
ENVASES	5341.679999999999
PAPEL-CARTON	3649.73
VIDRIO	2082.42
PUNTOS LIMPIOS	845.44
CARTON COMERCIAL	441.719999999999
RCD	432.045000000001

Total de toneladas por residuo en este distrito

Total Toneladas por Residuo en SAN BLAS-CANILLEJAS



Bibliografía

- Construcción del jar – Baeldung: <https://www.baeldung.com/kotlin/gradle-executable-jar>
- DataFrame de Kotlin – Kotlin: <https://kotlin.github.io/dataframe/overview.html>
- Serializa a XML con Serialization para jvm - <https://github.com/pdvrieze/xmlutil>
- Dokka Documentación Kotlin - <https://github.com/Kotlin/dokka>