Acceso a Datos- Práctica 01 – Residuos

Alejandro Sánchez Monzón y Mireya Sánchez Pinzón

ÍNDICE

Explicación de las clases.

Modelos

Bitácora: es el modelo (data class) que se ocupa de recopilar la información de cada ejecución que haga nuestro programa, mostrando la información detallada más abajo. Esta información se irá almacenando en diferentes objetos por cada ejecución y, posteriormente, se agrupará para imprimirla en un fichero de tipo XML.

VARIABLE	TIPO
id	UUID
instante	LocalDateTime
tipo	Enum TipoOpcion
exito	Boolean
tiempoEjecucion	Long

- → **Id**: es el identificador que tendrá cada una de las ejecuciones, se utiliza formato UUID debido a su contenido alfanumérico y la facilidad para generar identificadores aleatorios.
- → **Instante**: es el momento en el que se realiza la ejecución (la opción del programa seleccionada) siguiendo el formato ISO 8601. Utilizamos el tipo LocalDateTime para llamar así al método now() y tener el instante en el momento preciso.
- → **Tipo**: es una enum class la cual contiene tres valores (PARSER, RESUMEN_GLOBAL y RESUMEN_CIUDAD). Este tipo se parsea y se relaciona con la lectura del fichero utilizando su método .from().
- → **Éxito**: es un boolean que devolverá true si la ejecución se realiza con éxito y false si es, al contrario. Esta variable cambia dentro del controlador, dependiendo de si el programa lanza o no una excepción al ser ejecutado.
- → **Tiempo de ejecución**: de tipo Long, es el tiempo (en milisegundos) que el programa tarda en ejecutar los métodos que nosotros le indiquemos. Se consigue registrar dicho tiempo con el método measureTimeMillis{}.

Contenedor: es el modelo (data class) que se ocupa de recopilar la información del CSV acorde a los Contenedores en distintos objetos y guardar sus valores para su posterior utilización.

VARIABLE	TIPO
codigo	Int
tipo	Enum TipoContenedor
modelo	String
descripcion	String
cantidad	Int
lote	Int
distrito	String
barrio	String?
via	Enum TipoVia
nombre	String
numero	Int?
coordenadaX	Double
coordenadaY	Double
longitud	String
latitud	String
direccion	String

- → **Código**: código numérico de dicho contenedor en tipo Int.
- → **Tipo**: es una enum class que recoge todos los tipos de contenedores que hay en Madrid. Trae consigo un companion object que contiene un método .from() y que se ocupa de parsear dichos enums cuando se reciben como String al leer los CSV.
- → **Modelo**: tipo String que describe el modelo del contenedor dado.
- → **Descripción**: breve descripción del contenedor.
- → **Cantidad**: cantidad como tipo Int de contenedores que existen para ese registro.
- → **Lote**: número de lote como tipo Int, lo que ayuda a categorizar el contenedor.

- → **Distrito**: tipo String que describe el nombre del distrito en el cual se encuentra el contenedor.
- → **Barrio**: campo vacío, de ahí que utilicemos el tipo String?. Va ligado a un parser que cambia este cambio vacío a la palabra "null".
- → **Vía**: es una enum class que define el tipo de vía en la cual se encuentra el contenedor. Contiene tambien su companion object para parsear dicho enum.
- → **Nombre**: nombre de la vía como tipo String.
- → **Número**: número de la vía en la cual se encuentra el contenedor como tipo Int?, ya que algunos campos pueden encontrarse vacíos.
- → **Coordenada X**: valor de tipo Double para definir la posición exacta del contenedor.
- → **Coordenada Y**: valor de tipo Double para definir la posición exacta del contenedor.
- → **Longitud**: valor de tipo String para definir la posición exacta del contenedor.
- → **Latitud:** valor de tipo String para definir la posición exacta del contenedor.
- → **Dirección**: situación detellada del contenedor como tipo String.

Residuo: es el modelo (data class) que registra la información de los residuos los cuales leemos del fichero CSV. Y los guarda para su posterior utilización.

VARIABLE	TIPO
anio	Int
mes	String
lote	Int
tipo	Enum TipoResiduo
distrito	Int
nombreDistrito	String
toneladas	Double

- → **Año**: es la fecha a nivel anual de la que procede el registro, 2021 por defecto en todos los CSV como tipo Int.
- → **Mes**: mes exacto del registro del cual procede el registro como tipo String.
- → **Lote**: valor como tipo Int del lote al cual pertenece el Residuo. Útil a nivel identificativo.
- → **Tipo**: es una enum class que recoge los tipos de residuos y los almacena relacionándolos con su valor a String.
- → **Distrito**: es el código en tipo Int del código en el que se recogió dicho residuo.
- → **Nombre del distrito**: es el valor de tipo String que detalla el nombre del distrito.
- → **Toneladas**: es la cantidad en formato Double (posteriormente será necesario parsearla para quitar las ",") que revela la cantidad de residuos recogidos.

DTOs

ResiduoDTO: es un "Data Transfer Object", un tipo de objeto que se ocupa de definir los tipos de las variables de los objetos que tienen asociadas a tipos más simples, tales como String, Int, etc.

Este DTO se relaciona con el modelo Residuo.

VARIABLE	TIPO

anio	Int
mes	String
lote	Int
tipo	String
distrito	Int
nombreDistrito	String
toneladas	Double

Incluye a su vez las anotaciones de @Serializable, @SerialName y @XmlElement, parte de la librería de Kotlin Serialization y útil para proyectar su información en ficheros JSON y XML.

ContenedorDTO: es un "Data Transfer Object", un tipo de objeto que se ocupa de definir los tipos de las variables de los objetos que tienen asociadas a tipos más simples, tales como String, Int, etc.

Este DTO se relaciona con el modelo **Contenedor**.

VARIABLE	TIPO
codigo	Int
tipo	String
modelo	String
descripcion	String
cantidad	Int
lote	Int
distrito	String
barrio	String?
via	String
nombre	String
numero	Int?
coordenadaX	Double
coordenadaY	Double
longitud	String
latitud	String
direccion	String

Incluye a su vez las anotaciones de @Serializable, @SerialName y @XmlElement, parte de la librería de Kotlin Serialization y útil para proyectar su información en ficheros JSON y XML.

BitacoraDTO: es un "Data Transfer Object", un tipo de objeto que se ocupa de definir los tipos de las variables de los objetos que tienen asociadas a tipos más simples, tales como String, Int, etc.

Este DTO se relaciona con el modelo **Bitacora**.

VARIABLE	TIPO
id	String
instante	String
tipo	String
exito	Boolean

tiempoEjecucion	Long

Incluye a su vez las anotaciones de @Serializable, @SerialName y @XmlElement, parte de la librería de Kotlin Serialization y útil para proyectar su información en ficheros JSON y XML.

Mappers

Las clases o archivos de Kotlin categorizados como "Mappers" son, en nuestro proyecto, esos ficheros que contienen funciones útiles para convertir objetos de tipo DTO a Objetos de tipos complejos, y viceversa.

Estos ficheros implementan funciones como .toDTO() o como .to[Objeto cual sea]() que se encargan de convertir los valores determinados al tipo correspondiente, tal y como han sido definidos en las clases anteriormente explicadas.

A continuación, algunos ejemplos de dichos métodos:

```kotlin
fun Contenedor.toDTO(): ContenedorDTO {
    logger.info( msg: "Mapeando Contenedor a DTO.")
    return ContenedorDTO(
        codigo = this.codigo,
        tipo = this.tipo.toString(),
        modelo = this.modelo,
        descripcion = this.descripcion,
        cantidad = this.cantidad,
        lote = this.lote,
        distrito = this.distrito,
        barrio = this.barrio,
        via =this.via.toString(),
        nombre = this.nombre,
        numero = this.numero,
        coordenadaX = this.coordenadaX,
        coordenadaY = this.coordenadaY,
        longitud = this.longitud,
        latitud = this.latitud,
        direccion = this.direccion,

    )
```

```kotlin
fun ResiduoDTO.toResiduo(): Residuo {
    logger.info( msg: "Mapeando DTO a Residuo.")
    return Residuo (
        anio = this.anio,
        mes = this.mes,
        lote = this.lote,
        tipo = TipoResiduo.from(this.tipo),
        distrito = this.distrito,
        nombreDistrito = this.nombreDistrito,
        toneladas = this.toneladas
    )
}
```

Services

ServiceCSS: esta es una clase que contiene un único método, y es por la funionalidad de este, por lo que la clase existe. El método en cuestión es .writeCSS(). Este se encarga de guardar el código CSS necesario para nuestra página web en una variable de tipo String, crea un fichero llamado style.css (o lo sobreescribe, si existe) en la ruta que nosotros le introducimos por parámetros y por último escribe en ese fichero creado el contenido de la variable String.

De esta forma, quedaría un método tal que:

```
val ficheroHTML = File( pathname: directorio + File.separator + "style.css")
ficheroHTML.writeText(codigoCSS)
```

Donde “directorio” es la ruta donde se guardará dicho fichero (pasada por parámetros) y “codigoCSS” es la variable String en la cual se encuentra nuestro código para la hoja de estilos.

ServiceHTML: al igual que nuestra clase ServiceCSS, ServiceHTML también es una clase que contiene dos métodos de escritura de ficheros, en este caso, con extensión .html.

Los métodos de esta clase funcionan de la siguiente manera:

Ambos reciben por parámetros el directorio en el cual se guardará el archivo .html que creamos en su interior, el códigoHTML que escribiremos en dicho fichero, solo que esta vez el código viene el controlador, y no se genera dentro del método y, en el caso del resumen de un distrito, el distrito en cuestión.

Mostramos para cada función su código:

```
fun writeHTML(directorio: String, codigoHTML: String) {
    logger.info( msg: "Escribiendo HTML.")
    val ficheroHTML = File( pathname: directorio + File.separator + "resumen.html")
    ficheroHTML.writeText(codigoHTML)
}
```

```
fun writeHTMLDistrito(directorio: String, codigoHTML: String, distrito: String) {
    logger.info( msg: "Escribiendo HTML.")
    val ficheroHTML = File( pathname: directorio + File.separator + "resumen_$distrito.html")
    ficheroHTML.writeText(codigoHTML)
}
```

StorageCSV: en estas clases, agrupamos todos los métodos que se encargan de leer y escribir en formato CSV, sean para el modelo Contenedor, o el modelo Residuo. Su diferenciación va dentro de la ramificación que hemos realizado para cada método.

Los métodos de leer Residuos y Contenedores de los CSV y almacenarlos en sus respectivos objetos se hacen buscando, dentro de la ruta que le pasemos por parámetros, un CSV con un nombre determinado, modelo_residuos_2021.csv o contenedores_varios.csv.

La estructura de los métodos .**read**() es la misma en ambos casos:

1. Comenzamos creando un fichero que sea igual a la ruta que pasamos por parámetros.
2. Si este fichero termina con .csv, es el fichero que necesitamos, y procedemos a validarlo.
3. Si el fichero no termina con .csv, no es un archivo, sino una ruta, por lo que nosotros terminamos de completar la misma para que llegue al archivo.
4. Si el fichero pasa de forma exitosa las validaciones que se encuentran dentro del fichero "Validations" (explicado más adelante), procedemos a leer el CSV.
5. Ignoramos la primera línea, puesto que es la cabecera, y para cada separación guiada por ";" asignamos un campo, al cual, mapeando de forma correcta, creamos un DTO en el que introducimos la información del registro cada uno en la variable que le corresponde de forma ordenada. Es aquí donde vamos a parsear o castear las variables de tipo String a las que pide el DTO.
6. Debemos tener presente el saber cuándo lanzar, y de qué forma, las excepciones, para así parar el programa correctamente.

Aquí un ejemplo de un método de lectura:

```
fun readResiduo(directorio: String): List<ResiduoDTO> {
    logger.info( msg: "Leyendo CSV.")
    var ficheroResiduo = File(directorio)
    if (!directorio.endsWith( suffix: ".csv")) {
        ficheroResiduo = File( pathname: directorio + File.separator + "modelo_residuos_2021.csv")
    }

    if (ficheroResiduo.exists() && validateFileExtension(ficheroResiduo.toString())) {
        if (longitudCabeceraResiduos(
                ficheroResiduo.readLines().first()
            ) && separacionColumnas(ficheroResiduo.readLines().first()).equals(
                ficheroResiduo.readLines().first().split( ...delimiters: ";")
            )
        ) {
            return ficheroResiduo.readLines().drop( n: 1) List<String>
                .map { it.split( ...delimiters: ";") } List<List<String>>
                .map { campo ->
                    ResiduoDTO(
                        anio = campo[0].toInt(),
                        mes = campo[1],
                        lote = campo[2].toInt(),
                        tipo = campo[3],
                        distrito = campo[4].toInt(),
                        nombreDistrito = campo[5],
                        toneladas = parseDouble(campo[6]),
                    )
                }
        } else {
            throw IllegalArgumentException("El número de columnas no coincide con el deseado.");
        }

    } else {
        throw IllegalArgumentException("El fichero ${ficheroResiduo.absolutePath} no existe");
    }
} Pinzón, 04/10/2022 21:13 • Lectura y escritura CSV y JSON en situación idílica.
```

Por otro lado, el método de escritura (.**write**()) de cada uno de los Objetos, tiene una estructura más sencilla:

1. Inicialmente recibimos por parámetros la ruta del directorio donde irá el CSV y creamos un fichero que guardaremos en la misma.
2. En este fichero, escribimos en la primera línea la cabecera que deseemos, siempre respetando el orden de los valores acorde al CSV original.
3. Tras esto, habiendo recibido también una lista de DTOs por parámetros, iremos recorriendo la misma y añadiendo al fichero creado el valor de cada campo, separando estos por ";", en este caso.

A continuación, un ejemplo de escritura de un fichero CSV:

StorageJSON: al igual que la clase anterior, y compartiendo funcionalidades con la clase de StorageXML que explicaremos a continuación, la clase StorageJSON tiene como objetivo leer y escribir ficheros en formato .json, acorde al objeto que se esté trabajando en cada una de ellas, Residuo o Contenedor.

Para la lectura y escritura de estos métodos nos hemos ayudado de la librería de Serialization de Kotlin. Esta librería consigue que, con solo 4 líneas, y unos pocos parámetros, seamos capaces de generar o procesar ficheros enteros.

En el caso de la **escritura**:

1. Creamos un método write() que toma por parámetros la ruta en la cual crearemos el archivo, y una lista del objeto deseado como DTO.
2. El método se ocupará de crear el archivo con la extensión adecuada.
3. Mediante los métodos de la librería, generar un código JSON perfectamente estructurado utilizando la información de esa lista.

En el caso de la **lectura**:

1. Creamos un método read() que recibirá por parámetros la ruta en la que se encuentra el fichero a leer.
2. De esta ruta, sacamos el fichero.
3. Utilizando los métodos de la librería Serialization, retornamos una lista resultante de leer la información de dicho JSON según su estructura y almacenar los datos en objetos de tipo DTO.

```kotlin
fun readResiduo(directorio: String): List<ResiduoDTO> {
    logger.info( msg: "Leyendo JSON.")
    val ficheroResiduo = File( pathname: directorio + File.separator + "residuos_resultado_parser.json")
    return Json.decodeFromString(ficheroResiduo.readText())
}
```

StorageXML: exactamente la misma finalidad que la clase StorageXML, pero enfocado a ficheros de formto XML, y no JSON.

Para los métodos de esta clase utilizaremos también la librería de Serialization enfocada a XML.

Para las opciones de **escritura**:

1. Crearemos un método que reciba por parámetros la lista de DTOs que queremos escribir y el directorio en el cual generaremos el fichero .xml.
2. Una vez hecho esto, creamos con el método File() el archivo, haciendo especial hincapié en poner correctamente la extensión.
3. Posteriormente, creamos una variable “xml” que asigna el indent para cada campo.
4. Por último, escribimos en el fichero, con ayuda de los métodos de la librería, la lista de DTOs de nuestro objeto.

```kotlin
fun writeContenedor(directorio: String, contenedorDTO: List<ContenedorDTO>) {
    logger.info( msg: "Escribiendo XML.")
    val ficheroContenedor = File( pathname: directorio + File.separator + "contenedores_resultado_parse
    val xml = XML { indent = 4 }
    ficheroContenedor.writeText(xml.encodeToString(contenedorDTO))
}
```

Para las opciones de **lectura**:

1. Creamos un métodoq ue reciba por parámetros la ruta en la cual se encuentra el fichero XML a leer.
2. Una vez encontrado, creamos la variable “xml” tal y como hicimos con los métodos de leer y guardamos el fichero a leer en una variable.
3. Por último, utilizando los métodos de Serializable, leemos el fichero registro a registro siguiendo la estructura de XML y lo almacenamos en una lista de objetos de tipo DTO.

```kotlin
fun readResiduo(directorio: String): List<ResiduoDTO> {
    logger.info( msg: "Leyendo XML.")
    val xml = XML { indent = 4 }
    val ficheroResiduo = File( pathname: directorio + File.separator + "residuos_resultado_parser.xml")

    return xml.decodeFromString(ficheroResiduo.readText())
}
```

Inicialmente, el método de leer el fichero XML de bitácora no estaba implementado, sino que, en el fichero de escritura de mismo, utilizábamos el método .appendText() para sobrescribir los datos. Esto era una práctica errónea, ya que, al no regenerar el fichero, la etiqueta padre del XML desaparecía, convirtiéndolo en una estructura incompleta e imposible de ejecutar.

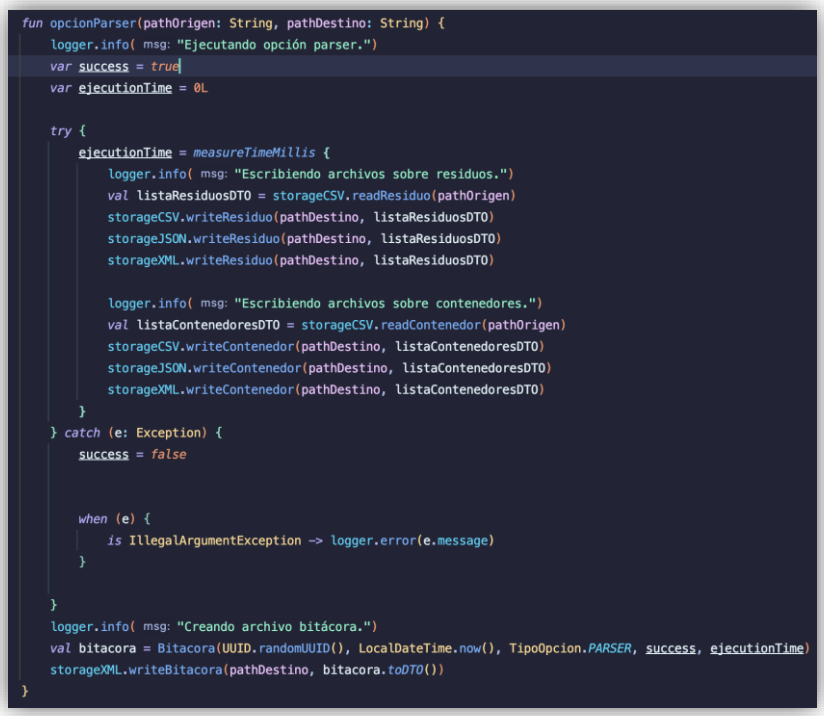
Para arreglar este problema, decidimos que la mejor solución era crear un método que leyese y almacenase las ejecuciones del proyecto que ya existía, que las guardase en una variable auxiliar, que metiese en esa variable la nueva ejecución, y que reescribiese esa lista. De esta forma, la estructura se corregía, siendo un XML funcional.

Controllers

En nuestro proyecto contamos con un único controlador que es capaz de controlar el flujo de la aplicación.

Los tres grandes y principales métodos son los que implementan cada opción que se nos pedía desarrollar.

Método opcionParser().



```
fun opcionParser(pathOrigen: String, pathDestino: String) {
    logger.info( msg: "Ejecutando opción parser.")
    var success = true
    var ejecutionTime = 0L

    try {
        ejecutionTime = measureTimeMillis {
            logger.info( msg: "Escribiendo archivos sobre residuos.")
            val listaResiduosDTO = storageCSV.readResiduo(pathOrigen)
            storageCSV.writeResiduo(pathDestino, listaResiduosDTO)
            storageJSON.writeResiduo(pathDestino, listaResiduosDTO)
            storageXML.writeResiduo(pathDestino, listaResiduosDTO)

            logger.info( msg: "Escribiendo archivos sobre contenedores.")
            val listaContenedoresDTO = storageCSV.readContenedor(pathOrigen)
            storageCSV.writeContenedor(pathDestino, listaContenedoresDTO)
            storageJSON.writeContenedor(pathDestino, listaContenedoresDTO)
            storageXML.writeContenedor(pathDestino, listaContenedoresDTO)
        }
    } catch (e: Exception) {
        success = false


        when (e) {
            is IllegalArgumentException -> logger.error(e.message)
        }

    }
    logger.info( msg: "Creando archivo bitácora.")
    val bitacora = Bitacora(UUID.randomUUID(), LocalDateTime.now(), TipoOpcion.PARSER, success, ejecutionTime)
    storageXML.writeBitacora(pathDestino, bitacora.toDTO())
}
```

A primera vista puede parecer muy enrevesado todo, pero realmente se ocupa de dos grandes bloques de funcionalidades:

1. Escribir los datos de los CSV originales en tres archivos de extensiones: “.csv”, “.json”, “.xml”.

Escribiremos en los archivos de las extensiones indicadas mediante los métodos de escritura cuya explicación detallada hemos desarrollado en el apartado anterior. Para escribir en esos archivos necesitamos la ruta donde se quieren almacenar, la cual la proporciona le usuario, y una lista de los datos que vamos a almacenar en formato DTO para que no haya ningún problema al escribir algún atributo de tipo no primario.

La lista de la que hablábamos la obtenemos a partir de leer el archivo original.

2. Escribir en un fichero “bitacora.xml” una lista de todas las ejecuciones del programa.

La forma de escribirlo es exactamente la misma que en la otra funcionalidad.

Lo que cambia aquí es que debemos de recolectar una serie de datos que obtendremos a partir de la propia ejecución del programa.

- UUID: este atributo lo definimos a partir de la función “randomUUID()” ya que al haber tantísimas combinaciones posibles es muy improbable que en algún momento salga la misma. De esta forma podremos atribuir a cada ejecución unos caracteres identificativos únicos.
- Tiempo inicio: El momento justo en el que se ha inicializado la ejecución de la opción elegida. Lo obtenemos a través de la función “now()”
- Opción elegida: Es fácil obtenerla según lo hemos enfocado nuestro proyecto porque ya nos hallamos en la propia elección.
- Éxito: Para saber si ha terminado como debería, hemos desarrollado una metodología que mediante el uso de un “try/catch” podemos cambiarle el valor a una variable de tipo booleeano. Si salta una excepción podemos confirmar que no ha ido como debería y reasignaremos el valor de la variable a la vez que controlamos la excepción.
- Tiempo Ejecución: lo hemos medido a través de encerrar la ejecución en un método que cuenta en milisegundos lo que hay en su interior.

Método opcionResumen().

Este método encierra cuatro bloques de funcionalidades:

1. Leer un archivo (.csv, .json o .xml) independientemente de la extensión y usar esta información para realizar unas consultas.

Esta funcionalidad la hemos afrontado de la misma forma que en la opción parser con el único cambio de que establecemos una prioridad de lectura.

Primero se intentará leer un archivo CSV, si no lo encuentra, un JSON y si tampoco lo encuentra un XML.

A partir de conseguir la lista DTO de la información deseada, obtenemos la información tipificada en base al modelo para poder hacer las consultas en base a unos datos correctamente definidos, y posteriormente lo pasamos a DataFrame ya que es la librería que hemos decidido utilizar.

2. Consultas y gráficos.

Hemos considerado que la forma más práctica y que más hace uso de código limpio es crear funciones independientes que nos devuelvan el resultado que esperamos, sobre todo para organizarnos mejor.

Más tarde se detallará el funcionamiento de cada consulta y cada gráfico.

3. Crear un archivo HTML con el resultado.

Par el HTML hemos decidido que la forma más ágil de hacerlo sería almacenando la plantilla de cómo queríamos que fuese nuestro HTML y haciendo llamadas a los métodos de cada consulta para incluir el valor del resultado.

Algo que hemos decidido tras algunos momentos de incertidumbre, ha sido añadir al final de cada consulta la función ".html()". Esta función lo que permite es darle un aspecto mucho más estructurado y más visual al resultado de la consulta. Estuvimos dubitativos ante este cambio ya que una pega que tiene esta función es que no muestra el resultado completo, muestra un número de filas específico y al final hace entender que el resultado continúa.

Elegimos finalmente añadirlo puesto que consideramos que lee objetivo principal del archivo HTML es permitir hacernos a la idea y comprender más fácilmente el mecanismo que sigue nuestro programa.

4. Escribir en un fichero "bitacora.xml" una lista de todas las ejecuciones del programa.

Funciona del mismo modo que la bitácora de la opción anterior.

Método opciónResumenDistrito().

Este método encierra las mismas funcionalidades que el anterior, con la única diferencia que permite al usuario introducir un distrito específico de Madrid sobre el que quiere hacer las consultas.

Las funcionalidades son las mismas en todos los casos excepto en las consultas, que como ya he mencionado se explicarán detalladamente más adelante.

Utils

Parsers: Es fichero de Kotlin que contiene las funciones encargadas de hacer un "parseo" de los valores de según que campos, a otros, que hacen más fácil su procesamiento y utilización. Los métodos del mismo ya han sido explicados a lo largo de la documentación.

Formatters: Este fichero de Kotlin contiene el método de formatear la fecha al formato español. Ya que este viene, por defecto, en formato estándar. Es necesario para reflejar correctamente la fecha de generación del HTML de las consultas y gráficas.

Validations: Es un fichero que contiene funciones encargadas de validar, como su propio nombre indica diferentes aspectos.

- Validar extensión de un archivo.

```
fun validateFileExtension(pathOrigen: String): Boolean {
    logger.info( msg: "Comprobando extensiones.")
    return pathOrigen.endsWith( suffix: ".csv") ||
            pathOrigen.endsWith( suffix: ".json") ||
            pathOrigen.endsWith( suffix: ".xml")
}
```

Esta función mediante una cadena de caracteres que hace referencia a la ruta del archivo en cuestión, comprueba si termina con alguna de las extensiones con las que trabajamos y si no es así no da como validada la extensión.

- Validar ruta.

```
fun validatePath(path: String): Boolean {
    logger.info( msg: "Comprobando ruta.")
    return File(path).exists()
}
```

Esta función se limita a comprobar que la cadena de caracteres que hace referencia a la ruta del archivo existe, si no es así no la da como válida.

- Validar longitud cabecera.

```
fun longitudCabeceraResiduos(cabecera: String): Boolean {
    return cabecera.split( ...delimiters: ";").size == 7
}
```

Tenemos dos funciones de este estilo. Esta en concreto hace referencia al CSV de Residuos del cual hemos decidido leer siempre todos los datos, aunque, por ahora, no los utilicemos. Para futuras ampliaciones del proyecto podría ser muy interesante.

Funciona de la siguiente forma, dada una cadena de caracteres que hace referencia a la cabecera del CSV en cuestión, creamos una lisat en la que cada elemento es el resultado de la separación por cada delimitador, que en nuestro caso es un punto y coma. Es decir, cada elemento de la lista es el nombre de una columna, si su tamaño es igual al esperado, 7 en este caso, es válida.

- Validar separación columnas.

```
fun separacionColumnas(cabecera: String): List<String> {
    return cabecera.split( ...delimiters: ";")
}
```

Esta función se limita a crear una lista que contenga en cada posición, el nombre de cada columna de la cabecera. Esto se hace de la misma forma que en el apartado anterior.

Main

Es la clase principal del programa, esta tiene la función “main”, la cual es la que hace que el programa se ejecute, y esta, a su vez, llama a un método llamado checkArgs(), el cual recibe por parámetros los argumentos que introduzcamos cuando ejecutemos el JAR.

Esta función, se ocupa de, por un lado, revisar que los argumentos que se están introduciendo son válidos y no son una fuente de excepciones que hagan “reventar” el programa.

Por otro lado, se encarga de analizar uno a uno los argumentos, y dependiendo de su patrón, su orden, el número de ellos, etc. Se encarga de llamar a los métodos del controlador que realizan cada opción, garantizando la seguridad de su ejecución sin cuelgues ni errores.

A continuación, la simplicidad del método **main**:

```
fun main(args: Array<String>) {
    logger.info( msg: "Iniciando programa ... ")
    checkArgs(args)
}
```

Tanto el StorageCSS como el método ggsave() de LetsPlot dentro de la generación de gráficos se ha diseñado minuciosamente para que, a la hora de generarte dichos archivos, lo haga de forma dinámica en la misma carpeta que te genera el HTML. Esto soluciona problemas de rutas al ejecutar la página web.

De esta forma, no llamamos a un CSS o a unas imágenes que luego el HTML debe buscar, sino que creamos una estructura web fácil de utilizar.

Transformación de formatos de la información.

Por el carácter de esta práctica es evidente que la información que procesamos va a sufrir cambios principalmente en el contenedor de dicha información y no en los datos en sí mismos.

Los primeros cambios con los que nos vamos a encontrar, y los más radicales, ocurrirán en la lectura y escritura de estos datos en ficheros externos. Porque como el propio enunciado indica, es necesario dar un formato ya que estandarizar los datos de una manera específica y universal es primordial.

Para dar el formato del que hablamos nuestro primer paso ha sido leer, en este caso, los archivos CSV indicados. En los cuales nos hemos encontrado algunos inconvenientes como: campos incompletos, decimales con comas bajas en lugar de puntos, mismos distritos escritos a veces con tildes y a veces sin ellas, etc.

Nuestro primer problema, campos incompletos.

```
fun parseNull(cadena: String): String {
    logger.info( msg: "Formateando campos vacíos.")
    var aux = ""
    if (cadena.isEmpty()) {
        aux = "null"
    }
    return aux
}
```

Lo hemos resuelto a través de la función anterior, con él lo que conseguimos es que, a partir de una cadena de caracteres, comprobamos si está vacía y si es así, en su lugar pondremos la palabra clave “null” para indicar que de este campo no se tiene información y así evitar posibles excepciones. Sin embargo, si en este campo ya contamos con una cadena de texto no vacía simplemente no la modificaremos.

El segundo problema, decimales con comas bajas.

```
fun parseDouble(cadena: String): Double {
    logger.info( msg: "Formateando decimales.")
    return cadena.replace( oldValue: ",",  newValue: ".").toDouble()
}
```

Hemos implementado la anterior función que nos permite reemplazar de una cadena de caracteres (un decimal pasado a cadena de caracteres), el carácter “,” por el carácter “.” ya que puesto que queremos almacenarlo en un tipo “Double” el estándar es utilizar un punto para diferenciar los valores decimales de los enteros.

El tercer problema, distritos a veces con tilde y a veces sin ella.

```
fun parseDistrito(distrito: String): String {
    var aux = ""

    if (distrito.equals("Chamberí")) {
        aux = distrito.replace( oldValue: "í",  newValue: "i")
    }
    if (distrito.uppercase().equals("Tetuán")) {
        aux = distrito.replace( oldValue: "á",  newValue: "a")
    }
    if (distrito.uppercase().equals("Chamartín")) {
        aux = distrito.replace( oldValue: "í",  newValue: "i")
    }
    if (distrito.uppercase().equals("Vicálvaro")) {
        aux = distrito.replace(distrito[4].toString(),  newValue: "a")
    }

    return aux.uppercase()
```

En esta función hemos reunido todos los distritos que ortográficamente puedan presentar una vocal con tilde. De esta forma a partir de una cadena de caracteres que representaría el distrito leído en

cuestión, comprobaríamos si se trata de uno de los distritos acentuados, si es así pero no se encuentra con tilde no ocurriría nada, por otra parte, si la llevase se remplazaría esa vocal acentuada por la vocal sin tilde.

Además de esto hemos aprovechado esta función para que el usuario no tenga que estar pendiente de si el distrito que busca se encuentra escrito en minúsculas, mayúsculas o una mezcla de las dos, ya que pasamos todos ellos a mayúsculas y así evitamos ese problema.

Para dar por concluidos los problemas, tenemos una pequeña excepción que se trata de las fechas. No lo hemos incluido como tal en el mismo grupo que anteriores puesto que realmente no es un problema en sí, si no que como comentábamos anteriormente, siempre que se pueda hay que estandarizar los datos y como para las fechas contamos con el formato ISO español, lo hemos utilizado.

```
fun dateFormatter(date: LocalDateTime): String {
    logger.info( msg: "Formateando fecha a la zona horaria de España.")
    return date.format(
        DateTimeFormatter
            .ofLocalizedDate(FormatStyle.FULL)
            .withLocale(Locale( language: "es",  country: "ES")))
}
```

En esta función dada una fecha tipificada con LocalDateTime le hemos aplicado un formato que facilita la propia clase a través del método “format()” en el que hemos especificado que queríamos la fecha lo más detallada posible (día de la semana, día del mes, mes y año) y, además bajo la zona horaria de España.

Una vez ya sabemos bajo que patrones vamos a estandarizar los datos podemos comenzar con la lectura.

Como es lógico, al extraer los datos de un fichero CSV lo que obtenemos realmente son cadenas de caracteres, y aunque esto está bastante bien para almacenar información no lo está tanto para operar con ella.

Es por ello que nos vamos a encontrar una nueva forma de encapsular la información “DTOs” y “Modelos”.

Los DTO o bien, Objetos de Transferencia de Datos, nos van a servir, como un intermediario para llegar a lo que nosotros realmente queremos, los Modelos. Estos intermediarios cumplen una función primordial en ser los primeros que clasifican mediante atributos los datos, dotándolos además de una primera tipificación que más adelante seremos capaces de afinar. Vamos a trabajar especialmente con ellos en operaciones tales como entrada y salida de almacenamiento externo.

Los Modelos, son la forma de tipificación final de nuestros datos. Están diseñados específicamente para que nos faciliten y seamos capaces de operar con los datos como necesitamos.

Sabemos cómo se crean los Objetos de Transferencia de Datos, a partir de la lectura de un fichero en este caso. Pero, ¿cómo podremos pasar de ellos a modelos con los que podamos operar y viceversa? Mediante unas funciones que crearemos para mapear atributo a atributo.

```kotlin
fun ContenedorDTO.toContenedor(): Contenedor {
    logger.info( msg: "Mapeando DTO a Contenedor.")
    return Contenedor(
        codigo = this.codigo,
        tipo = TipoContenedor.from(this.tipo),
        modelo = this.modelo,
        descripcion = this.descripcion,
        cantidad = this.cantidad,
        lote = this.lote,
        distrito = this.distrito,
        barrio = this.barrio,
        via = TipoVia.from(this.via),
        nombre = this.nombre,
        numero = this.numero,
        coordenadaX = this.coordenadaX,
        coordenadaY = this.coordenadaY,
        longitud = this.longitud,
        latitud = this.latitud,
        direccion = this.direccion,
    )
}
```

Con la función anterior obtendremos el Modelo de un DTO igualando cada atributo y formateándolo de forma más concreta al atributo del Modelo.

Por lo tanto, ahora cabría explicar cómo hacer la operación inversa. Sin embargo, en nuestro proyecto lo hemos planteado de forma que no sea necesaria esta función. Mediante la lista que nos devuelve la función de lectura la utilizamos en la función de escritura parra escribirla tal cual. Esto lo podemos hacer ya que en nuestro proyecto los datos del fichero jamás cambian, simplemente los consultamos, si esto no fuese así de ninguna forma podríamos enfocarlo de la misma manera y deberíamos implementar la función de la operación inversa.

Realización de consultas.

Todo Madrid:

Número de contenedores de cada tipo que hay en cada distrito

```kotlin
private fun numeroContenedoresPorTipoPorDistrito(listaContenedores: DataFrame<Contenedor>): String {
    logger.info( msg: "Consultando el número de contenedores de cada tipo que hay en cada distrito.")
    return listaContenedores
        .groupBy( ...cols: "distrito", "tipo") GroupBy<Contenedor, Contenedor>
        .aggregate { this: AggregateGroupedDsl<Contenedor>  it: AggregateGroupedDsl<Contenedor>
            sum( ...columns: "cantidad") into "Total"
        } DataFrame<Contenedor>
        .sortBy( ...cols: "distrito").html()
}
```

En esta consulta, hemos agrupado por distritos y tipos de residuos, y hemos llamado a una agregación de ambos que añade una suma de las cantidades de cada tipo de contenedor por distritos, añadiéndolo a una columna llamada “Total”. Tras esto, ordenamos dicha consulta por distritos y le introducimos el método .html() para que se vea un resultado más visual.

Media de contenedores de cada tipo que hay en cada distrito.

```
private fun mediaContenedoresPorTipoPorDistrito(listaContenedores: DataFrame<Contenedor>): String {
    logger.info( msg: "Consultando media de contenedores de cada tipo que hay en cada distrito.")
    return listaContenedores
        .groupBy( ...cols: "distrito", "tipo") GroupBy<Contenedor, Contenedor>
        .aggregate { this: AggregateGroupedDsl<Contenedor>  it: AggregateGroupedDsl<Contenedor>
            mean( ...columns: "cantidad") into "Media"
        } DataFrame<Contenedor>
        .sortBy( ...cols: "distrito").html()
}
```

En esta consulta, agrupamos la lista por distritos y tipos de contenedores de nuevo, pero esta vez, realizamos una media de las veces que sale cada tipo en cada distrito (cantidad) y lo guardamos en una columna llamada “Media”. Por último, la ordenamos por distritos y llamamos al método .html().

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

```
private fun mediaToneladasAnualesPorTipoResiuoPorDistrito(listaResiduos: DataFrame<Residuo>): String {
    logger.info( msg: "Consultando media de toneladas anuales de recogidas por cada tipo de basura agrupadas por distrito.")
    return listaResiduos
        .groupBy( ...cols: "nombreDistrito", "tipo", "anio") GroupBy<Residuo, Residuo>
        .aggregate { this: AggregateGroupedDsl<Residuo>  it: AggregateGroupedDsl<Residuo>
            mean( ...columns: "toneladas") into "Media"
        } DataFrame<Residuo>
        .sortBy( ...cols: "nombreDistrito").html()
}
```

En esta consulta, agrupamos por el nombre del distrito, el tipo de residuo y el año, que será siempre igual a 2021. Una vez hecho esto llamamos a la agregación de esta agrupación y sacamos la media de toneladas de cada tipo de residuo, creando una columna llamada “Media”. Por último, ordenamos por distritos.

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

```
private fun estadisticasToneladasAnualesPorTipoPorDistrito(listaResiduos: DataFrame<Residuo>): String {
    logger.info( msg: "Consultando máximo, mínimo, media y desviación de toneladas anuales de recogidas por cada tipo de basura agrupadas por distrito.")
    return listaResiduos
        .groupBy( ...cols: "nombreDistrito", "tipo", "anio") GroupBy<Residuo, Residuo>
        .aggregate { this: AggregateGroupedDsl<Residuo>  it: AggregateGroupedDsl<Residuo>
            mean( ...columns: "toneladas") into "Media"
            min( ...columns: "toneladas") into "Mínimo"
            max( ...columns: "toneladas") into "Máximo"
            std( ...columns: "toneladas").toString() into "Desviación" ^aggregate
        } DataFrame<Residuo>
        .sortBy( ...cols: "nombreDistrito").html()
}
```

En esta consulta, agrupamos la lista como en la anterior, pero esta vez, haremos no solo la media sino también el máximo, mínimo y desviación (este último no es un número entero, por lo que debemos parsearlo). Guardamos cada valor en una columna, ordenamos por distritos y embellecemos la consulta.

Suma de todo lo recogido en un año por distrito.

```
private fun cantidadResiduosAnualPorDistrito(listaResiduos: DataFrame<Residuo>): String {
    logger.info( msg: "Consultando suma de to do lo recogido en un año por distrito.")
    return listaResiduos
        .filter { it["anio"] == 2021 } DataFrame<Residuo>
        .groupBy( ...cols: "nombreDistrito") GroupBy<Residuo, Residuo>
        .aggregate { this: AggregateGroupedDsl<Residuo>  it: AggregateGroupedDsl<Residuo>
            sum( ...columns: "toneladas") into "TotalAnual"
        } DataFrame<Residuo>
        .sortBy( ...cols: "nombreDistrito").html()
}
```

En esta consulta, la cual debe ser del año 2021 únicamente (por eso filtramos), agruparemos por distritos y sumaremos todas las toneladas recogidas en ese año por cada uno de los distritos. Al final, ordenaremos dicha consulta y la embelleceremos.

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

```
private fun cantidadResiduoPorTipoPorDistrito(listaResiduos: DataFrame<Residuo>): String {
    logger.info( msg: "Consultando por cada distrito obtener para cada tipo de residuo la cantidad recogida.")
    return listaResiduos
        .groupBy( ...cols: "nombreDistrito", "tipo") GroupBy<Residuo, Residuo>
        .aggregate { this: AggregateGroupedDsl<Residuo>  it: AggregateGroupedDsl<Residuo>
            sum( ...columns: "toneladas") into "TotalResiduo"
        } DataFrame<Residuo>
        .sortBy( ...cols: "nombreDistrito").html()
}
```

En esta consulta, agruparemos la lista por distritos y tipos de residuos. Una vez hecho esto, realizaremos una suma de las toneladas recogidas para cada tipo en cada distrito, y ordenaremos la misma por distritos, creando a su vez una columna para guardar los valores resultantes.

Distrito concreto:

Número de contenedores de cada tipo que hay en este distrito.

```
private fun numeroContenedoresPorTipoEnDistrito(listaContenedores: DataFrame<Contenedor>, distrito: String): String {
    logger.info( msg: "Consultando número de contenedores de cada tipo que hay en $distrito")

    return listaContenedores
        .filter { it["distrito"] == parseDistrito(distrito)} DataFrame<Contenedor>
        .groupBy( ...cols: "distrito", "tipo") GroupBy<Contenedor, Contenedor>
        .aggregate { this: AggregateGroupedDsl<Contenedor>  it: AggregateGroupedDsl<Contenedor>
            sum( ...columns: "cantidad") into "NumeroContenedores"
        } DataFrame<Contenedor>
        .sortBy( ...cols: "distrito").html()
}
```

En esta consulta, lo primero que debemos hacer es filtrar para obtener los datos solo del distrito requerido, una vez hecho esto, agruparemos la información por el tipo de contenedor y, de aquí, sacaremos la suma de cuantos contenedores hay de cada tipo.

Total, de toneladas recogidas en ese distrito por residuo.

```
private fun cantidadToneladasPorResiduoEnDistrito(listaResiduos: DataFrame<Residuo>, distrito: String): String {
    logger.info( msg: "Consultando total de toneladas recogidas en $distrito por residuo.")
    return listaResiduos
        .filter { it["nombreDistrito"] == distrito } DataFrame<Residuo>
        .groupBy( ...cols: "tipo") GroupBy<Residuo, Residuo>
        .aggregate { this: AggregateGroupedDsl<Residuo>  it: AggregateGroupedDsl<Residuo>
            sum( ...columns: "toneladas") into "Toneladas"
        } DataFrame<Residuo>
        .html()
}
```

En esta consulta, lo primero que debemos hacer es filtrar para obtener los datos solo del distrito requerido, una vez hecho esto, agruparemos por tipo de residuos y sumaremos las toneladas recogidas para cada tipo en ese distrito, agrupando los datos en una columna llamada “Toneladas”.

Máximo, mínimo, media y desviación por mes por residuo en dicho distrito.

```
private fun estadisticasMensualesPorTipoEnDistrito(listaResiduos: DataFrame<Residuo>, distrito: String): String {
    logger.info( msg: "Consultando máximo, mínimo , media y desviación por mes por residuo en $distrito.")
    return listaResiduos
        .filter { it["nombreDistrito"] == distrito } DataFrame<Residuo>
        .groupBy( ...cols: "nombreDistrito", "tipo", "anio") GroupBy<Residuo, Residuo>
        .aggregate { this: AggregateGroupedDsl<Residuo>  it: AggregateGroupedDsl<Residuo>
            mean( ...columns: "toneladas") into "Media"
            min( ...columns: "toneladas") into "Mínimo"
            max( ...columns: "toneladas") into "Máximo"
            std( ...columns: "toneladas").toString() into "Desviación" ^aggregate
        } DataFrame<Residuo>
        .html()
}
```

En esta consulta, lo primero que debemos hacer es filtrar para obtener los datos solo del distrito requerido, una vez hecho esto, agruparemos por tipo de residuos y por año (2021, ya que no hay registros para otro año). Tras esto, haremos una agregación y operaremos sacando la media, mínimo y desviación de las toneladas de cada residuo ene se año, en ese distrito, creando una columna para cada operación.

Gráficos.

Todo Madrid:

Gráfico con el total de contenedores por distrito.

```
private fun graficoTotalContenedoresDistrito(listaContenedores: DataFrame<Contenedor>, pathDestino: String) {
    logger.info( msg: "Creando gráfico con el total de contenedores por distrito. ")
    val res = listaContenedores
        .groupBy( ...cols: "distrito", "tipo") GroupBy<Contenedor, Contenedor>
        .aggregate { this: AggregateGroupedDsl<Contenedor>  it: AggregateGroupedDsl<Contenedor>
            sum( ...columns: "cantidad") into "TotalContenedores"
        } DataFrame<Contenedor>
        .toMap()

    val fig: Plot = letsPlot(data = res) + geomBar(
        stat = identity,
        alpha = 0.8,
        fill = Color.BLACK,
        color = Color.ORANGE
    ) { this: BarMapping
        x = "distrito"
        y = "TotalContenedores"
    } + labs(
        x = "Distrito",
        y = "Total de contenedores",
        title = "Total de contenedores por distrito."
    )

    val path = pathDestino + File.separator + "images"
    if (!Paths.get(path).exists()) {
        Files.createDirectory(Paths.get( first: pathDestino + File.separator + "images" + File.separator))
    }

    ggsave(fig, path = path + File.separator, filename = "contenedores_distritos.png")
```

En esta gráfica, hacemos una agrupación por distritos y por tipos de contenedores de la lista, y posteriormente una agregación, sumando la cantidad de contenedores de cada tipo en cada distrito.

Esta consulta debemos pasarla a un mapa para trabajar con ella.

Posteriormente a esto, creamos una variable que contendrá los datos resultantes de la consulta, la cual será de un gráfico de barras negras y bordes naranjas.

Para lo comentado anteriormente con el problema de las rutas del HTML, crearemos una ruta donde introduciremos la gráfica, y esta ruta con su contenido será movido a esa ruta donde se genere el resumen.html. Esto será así con todas las gráficas.

Gráfico de media de toneladas mensuales de recogida de basura por distritos.

```
private fun graficoMediaToneladasDistrito(listaResiduos: DataFrame<Residuo>, pathDestino: String) {
    logger.info( msg: "Creando gráfico de media de toneladas mensuales de recogida de basura por distrito.")
    val res = listaResiduos
        .groupBy( ...cols: "nombreDistrito", "mes") GroupBy<Residuo, Residuo>
        .aggregate { this: AggregateGroupedDsl<Residuo>  it: AggregateGroupedDsl<Residuo>
            mean( ...columns: "toneladas") into "media"
        } DataFrame<Residuo>
        .toMap()

    val fig: Plot = letsPlot(data = res) + geomPoint(
        stat = identity,
        alpha = 0.8,
        fill = Color.BLACK,
        color = Color.BLACK
    ) { this: PointMapping
        x = "nombreDistrito"
        y = "media"
    } + labs(
        x = "Distrito",
        y = "Media de basura recogida.",
        title = "Media de toneladas mensuales de recogida de basura por distrito."
    )

    val path = pathDestino + File.separator + "images"
    if (!Paths.get(path).exists()) {
        Files.createDirectory(Paths.get( first: pathDestino + File.separator + "images" + File.separator))
    }

    ggsave(fig, path = path + File.separator, filename = "media_toneladas_distrito.png")
```

En esta gráfica, aremos una agrupación de la lista por el distrito y el mes, realizando la media de las toneladas recogidas para cada uno de ellos en la fecha marcada.

Para esta gráfica utilizaremos un sistema de puntos, el cual agrupa los valores de forma progresiva, dejando ver de forma clara los puntos (de color negro) en los picos más altos y más bajos.

Distrito concreto:

Gráfico con el total de toneladas por residuo en ese distrito.

```
private fun graficoTotalToneladasResiduoDistrito(listaResiduos: DataFrame<Residuo>, distrito: String, pathDestino: String) {
    logger.info( msg: "Creando gráfico con el total de toneladas por residuo en $distrito.")
    val res = listaResiduos
        .filter { it["nombreDistrito"] == distrito } DataFrame<Residuo>
        .groupBy( ...cols: "tipo", "toneladas") GroupBy<Residuo, Residuo>
        .aggregate { this: AggregateGroupedDsl<Residuo>  it: AggregateGroupedDsl<Residuo>
            count() into "total"
        } DataFrame<Residuo>
        .toMap()

    val fig: Plot = letsPlot(data = res) + geomPoint(
        stat = identity,
        alpha = 0.8,
        fill = Color.BLACK,
        color = Color.BLACK,
    ) { this: PointMapping
        x = "tipo"
        y = "toneladas"
    } + labs(
        x = "Tipo de residuo",
        y = "Toneladas",
        title = "Total de toneladas por residuo en $distrito"
    )

    val path = pathDestino + File.separator + "images"
    if (!Paths.get(path).exists()) {
        Files.createDirectory(Paths.get( first: pathDestino + File.separator + "images" + File.separator))
    }
```

En esta gráfica realizaremos un filtro con el distrito deseado, de este, agruparemos por tipo de residuos y toneladas recogidas y sacaremos un total, guardando dicha variable en un campo llamado "total". Posteriormente, realizaremos una gráfica de puntos la cual mostrará los picos más altos y bajos, incluidos la progresión, por distritos y cantidades.

Gráfica del máximo, mínimo y media por meses en dicho distrito.

```
private fun graficoMaxMinMediaPorMeses(listaResiduos: DataFrame<Residuo>, distrito: String, pathDestino: String) {
    logger.info( msg: "Creando gráfica del máximo, mínimo y media por meses en $distrito.")
    val res = listaResiduos.filter { it["nombreDistrito"] == distrito } DataFrame<Residuo>
        .groupBy( ...cols: "nombreDistrito", "mes") GroupBy<Residuo, Residuo>
        .aggregate { this: AggregateGroupedDsl<Residuo>  it: AggregateGroupedDsl<Residuo>
            max( ...columns: "toneladas") into "Máximo"
            min( ...columns: "toneladas") into "Mínimo"
            mean( ...columns: "toneladas") into "Media" ^aggregate
        } DataFrame<Residuo>
        .toMap()

    val fig: Plot = letsPlot(data = res) + geomBar(
        stat = identity,
        alpha = 0.8,
        fill = Color.GREEN,
        color = Color.WHITE
    ) { this: BarMapping
        x = "mes"
        y = "Máximo"
    } + geomBar(
        stat = identity,
        alpha = 0.8,
        fill = Color.LIGHT_YELLOW,
        color = Color.WHITE
    ) { this: BarMapping
        x = "mes"
        y = "Media"
    } + geomBar(
        stat = identity,
```

Para este gráfico hemos filtrado una vez más por el distrito indicado, y hemos realizado una agrupación por meses, del cual sacaremos las operaciones de máximo, mínimo y media de dicho mes para las toneladas recogidas.

Representaremos la gráfica como un gráfico de barras en el cual cada valor tomará un color. Siendo las medias naranjas, los máximos, verdes, y los mínimos, rojos. Es importante ordenar bien esta composición de barras para no superponer datos, dejando al fondo los máximos (valores más altos) y al frente los mínimos (valores más bajos).

De esta forma, la gráfica se verá tal que un gráfico con tres barras por posición, cada una de un color, mostrando de forma visual la diferenciación de los datos.

Justificación tecnológica.

Kotlin

Nos hemos decantado por utilizar este lenguaje por muchos motivos, entre ellos:

- El código es más simple y necesita menos líneas para realizar su función. Esto reduce el margen de error ya que es más evitable perderse en él por lo tanto y facilitará su trabajo.
- En algunos casos se ha convertido en una evolución de Java.

- Tiene muchas funciones que aún las versiones más nuevas de Java no son capaces de implementar, lo que hace que el desarrollo del programa sea más eficaz y llevadero.

DataFrame

Elegimos usar DataFrame para nuestras consultas ya que a pesar de que nuestros primeros intentos fuesen con las propias colecciones de Kotlin, nos resultó más enrevesado manejarlas a mano. DataFrame es capaz de agrupar de forma automática y gestionar la información de una forma mucho más dinámica que hacerlo con los streams de Kotlin, algo que se hace bastante tedioso a largo plazo.

Una de las ventajas que vimos de DataFrame fue que en menos líneas de código y de forma más intuitiva éramos capaces de manejar la información de manera infinitamente más ágil y así permitir el desarrollo de futuras ampliaciones muy eficazmente.

LetsPlots

Escogimos LetsPlots porque a primera vista nos pareció la forma más sencilla de representar gráficas.

LetsPlot tiene bastante afinidad con DataFrame y la combinación de ambas hace que su uso sea mucho más sencillo, eso nos llevó también a, una vez que sabíamos que librería utilizaríamos para las consultas, decantarlos por la librería para las gráficas.

Sin embargo, cuando empezamos a indagar un poco más descubrimos la infinidad de diseños que podíamos crear simplemente conociéndolos.

Cabe destacar que como el uso que dimos a las gráficas que obtuvimos era para el HTML nos pareció una de las mejores soluciones para fomentar lo que ya hemos mencionado a lo largo del proyecto: que el HTML fuese muy visual, estructurado y atractivo visualmente.

Serialization

Creemos que elegir esta librería fue la decisión más sencilla y más unánime de este proyecto.

Esto se debe a que nos llamó mucho la atención que simplemente con ella pudiésemos hacer tantas cosas. Esto lo que provocó en nosotros fue bastante soltura en este aspecto a medida que avanzaba el proyecto y que fuésemos muchísimo más resolutivos a la hora de afrontar errores y solventarlos.

Dokka

Inicialmente nos decantamos por KDoc pero al ver la cantidad de errores que nos daba al construir el proyecto empezamos a buscar otras librerías. Esto se debía a que no era compatible con otras dependencias.

En la búsqueda de otra librería similar dimos con Dokka que cumplía todos los requisitos que buscábamos para completar nuestra documentación y era muy ágil de usar.

En nuestro proyecto, Dokka está configurado desde Gradle, por lo que, para ejecutarlo, y poder generar un archivo JDOC o KDOC, es necesario acceder a la pestaña de Gradle y ejecutar la tarea indicada desde ahí.